Collection Ressources Informatiques

Algorithmique

Techniques fondamentales de programmation (avec des exemples en PHP)




Sébastien ROHAUT



INFORMATIQUE TECHNIQUE



Algorithmique



Techniques fondamentales de programmation (exemples en PHP)

Sébastien ROHAUT



Résumé

Ce livre sur l'algorithmique s'adresse à toute personne désireuse de maîtriser les **bases essentielles de la programmation**. Pour apprendre à programmer, il faut d'abord comprendre ce qu'est vraiment un ordinateur, comment il fonctionne et surtout comment il peut faire fonctionner des programmes, comment il manipule et stocke les données et les instructions, quelle est sa logique. Alors, au fur et à mesure, le reste devient évidence : variables, tests, conditions, boucles, tableaux, fonctions, fichiers, **jusqu'aux notions avancées** comme les pointeurs et les objets.
Dans ce livre, le langage algorithmique (ou la syntaxe du pseudo-code des algorithmes) reprend celui couramment utilisé dans les **écoles d'informatique** et dans les formations comme les **BTS, DUT, classes préparatoires, premières années d'ingénierie** à qui ce livre est en partie destiné et conseillé.
Une fois les notions de base acquises, le lecteur trouvera dans ce livre de quoi évoluer vers des notions plus avancées : deux chapitres, l'un sur les pointeurs et les références, l'autre sur les objets, ouvrent les portes de la programmation dans des langages évolués et puissants comme le **C**, le **C++**, **JAVA** et **PHP**.
La plupart des algorithmes de ce livre sont traduits en PHP et les sources, directement utilisables, sont disponibles en téléchargement sur cette page.

L'auteur

Sébastien ROHAUT a débuté comme Ingénieur de développement en C++. Aujourd'hui Ingénieur Système il intervient sur des missions régulières pour de grands comptes et continue d'enseigner le développement à des classes d'ingénieur et masters MTIC.

Introduction

Pourquoi apprendre à programmer ? Avez-vous, comme l'auteur, disposé au début de la micro-informatique d'un ordinateur où il fallait programmer soi-même des jeux ou outils, ou saisir des dizaines de pages de lignes de programmation ? Avez-vous besoin, durant vos études, de maîtriser les techniques fondamentales de programmation pour passer votre diplôme? Êtes-vous un professionnel ou un autodidacte passionné qui veut encore en savoir davantage ? Est-ce une nouvelle étape de votre carrière professionnelle où n'étant pas informaticien vous êtes amené à programmer des macros ou des scripts complexes ? Quelle raison encore trouver ? Si vous répondez oui à l'une des ces questions, mais aussi aux dizaines d'autres qu'il serait possible de poser, alors oui, vous devez apprendre à programmer. Apprendre à programmer, c'est enfin savoir comment font les autres pour créer de superbes logiciels, c'est savoir à terme comment les créer soi-même et les développer.

Comment apprendre à programmer ? On ne s'improvise pas programmeur. C'est un métier et comme tout métier, cela s'apprend. Dans les écoles, des professeurs enseignants pour des classes de BTS, DUT, DEUG, classes préparatoires, etc., sont spécialisés dans l'apprentissage des notions fondamentales de programmation. Les autodidactes se plongent dans des livres, des sites Internet, dans la documentation en ligne des langages, pour apprendre ces notions. L'ensemble de ces notions c'est l'algorithmique.

Ce livre reprend les notions essentielles, fondamentales, de la programmation. Pour apprendre à programmer, il faut d'abord comprendre ce qu'est vraiment un ordinateur, comment il fonctionne et surtout comment il peut faire fonctionner des programmes, comment il manipule et stocke les données et les instructions, quelle est sa logique. Alors, au fur et à mesure, le reste coule de source comme une évidence : variables, tests, conditions, boucles, tableaux, fonctions, fichiers, jusqu'aux notions avancées comme les pointeurs et les objets.

Le formalisme algorithmique, (la syntaxe du langage algorithmique ou pseudocode) reprend celui couramment utilisé dans les écoles d'informatique et dans les formations comme les BTS, DUT, premières années d'ingénierie, à qui ce livre est en partie destiné et conseillé. Il existe plusieurs variantes utilisées, selon le professeur, le langage d'origine. Celui présenté ici a l'avantage d'être dans un français très explicite "tant que, jusqu'à, pour chaque, afficher, saisir, etc.". Leur lecture ne nécessite aucune connaissance préalable de termes trop techniques.

Ce livre ne fait pas qu'aborder les notions basiques. Deux chapitres, l'un sur les pointeurs et les références, l'autre sur les objets, ouvrent les portes de la programmation dans des langages évolués et puissants comme le C, le C++, Java et PHP. D'ailleurs, presque tous les algorithmes de ce livre sont implémentés en PHP. Les sources directement utilisables sont disponibles en téléchargement sur le site des Éditions ENI.

Les fondements de l'informatique

1. Architecture de Von Neumann

Un ordinateur est un ensemble de circuits électroniques permettant de manipuler des informations qu'on appelle des données et capable de faire "tourner" des programmes, c'est-à-dire une suite ou séquence d'instructions programmées à l'avance et qu'il va dérouler du début à la fin dans le but d'obtenir des résultats. Pour comprendre comment un ordinateur peut dérouler un programme, il faut étudier un peu plus en détail son fonctionnement.

C'est **Von Neumann** qui a défini en 1944 l'architecture des ordinateurs modernes encore largement utilisée aujourd'hui (avec des variantes cependant). L'architecture de Von Neumann (issue des travaux de Turing dont il sera question plus loin) décompose l'ordinateur en quatre parties distinctes :

- L'**Unité Arithmétique et Logique** UAL (ALU en anglais) est l'organe de l'ordinateur qui exécute les calculs : additions, soustractions, multiplications, divisions, modulos, gestion des signes (positif, négatif), opérations logiques (booléenne), comparaisons, parfois rotations et décalages de valeurs (toujours dans le cadre d'une logique booléenne). Il existe des UAL spécialisées dans les nombres à virgule flottante, d'autres dans des traitements complexes comme les logarithmes, les inversions, les racines, les vecteurs, les calculs trigonométriques, etc. Certaines documentations lui rajoutent quelques registres (petites cases mémoires intégrées à l'UAL) et lui donnent le nom de **processeur** (CPU).

- L'**Unité de Contrôle** UC (CU en anglais), à ne pas confondre avec Unité Centrale, contrôle le séquençage des opérations, autrement dit le déroulement du programme. Elle prend ses instructions dans la mémoire et donne ses ordres à l'UAL. Les résultats retournés peuvent influer sur le séquençage. L'UC passe alors à l'instruction suivante ou à une autre instruction telle que le programme lui ordonne d'effectuer.

- La **mémoire** peut être décrite comme une suite de petites cases numérotées, chaque case pouvant contenir une petite information (petite dans le sens où la taille de chaque case est fixe). Cette information peut être une instruction ou un morceau d'instruction du programme (une instruction peut occuper plusieurs cases) ou une donnée (nombre, caractère, ou morceau de ceux-ci). C'est l'UC qui a comme rôle central de contrôler l'accès à la mémoire pour le programme et les données. Chaque numéro de case est appelé une adresse. Pour accéder à la mémoire, il suffit de connaître son adresse. Les instructions du programme pour l'UC et les données pour l'UAL sont placées dans des zones différentes de la même mémoire physique.

- **Les Entrées/Sorties** E/S (I/O en anglais) permettent de communiquer avec le monde extérieur et donc vous : ce peut être un clavier pour entrer les données, et un écran pour afficher les résultats. Il permet à l'ordinateur d'être interactif.

Les instructions du programme sont présentes dans la mémoire. L'unité de contrôle va prendre la première instruction du programme et l'exécuter. Si l'instruction est par exemple d'additionner deux nombres, elle va demander à l'UAL de prendre ces deux nombres en mémoire et de les additionner et éventuellement de placer le résultat dans une nouvelle case. Puis l'UC passe à l'instruction suivante. Si elle consiste à afficher ce résultat, alors l'UC va lire le contenu de la mémoire à l'adresse où est placé le résultat, puis va envoyer le résultat via le composant d'E/S adéquat. Et ainsi de suite. Au final le déroulement d'un programme au sein de l'ordinateur est le suivant :

- l'UC extrait une instruction de la mémoire,
- analyse l'instruction,
- recherche en mémoire les données concernées par l'instruction,
- déclenche l'opération adéquate sur l'ALU ou l'E/S,
- range le résultat dans la mémoire.

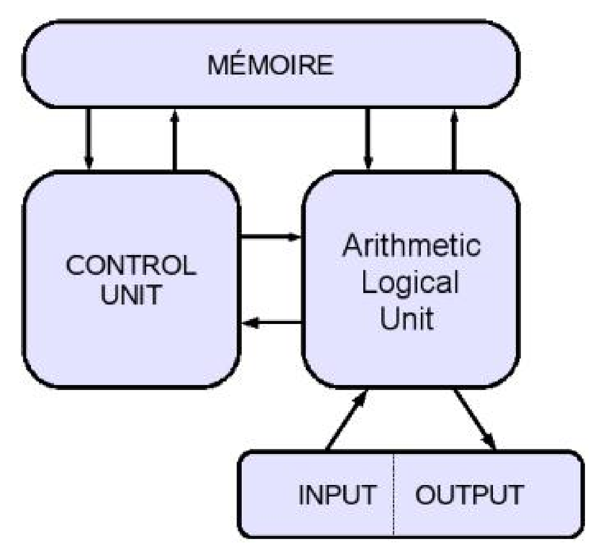


Von Neumann, père des ordinateurs actuels

Si vous ouvrez le capot de votre ordinateur, vous y verrez une grande quantité de cartes, composants, câbles, et même des organes mécaniques (lecteurs de disques durs, cd et disquette). Un programme que vous allez écrire et dérouler ne s'exécute pourtant que dans un seul endroit : le microprocesseur. Le microprocesseur de votre ordinateur est une puce facilement reconnaissable car c'est souvent la plus grosse, celle qui dispose du plus de pattes et est généralement surmontée d'un gros bloc d'aluminium ou de cuivre accompagné d'un ventilateur pour le refroidir. Il contient l'UAL, l'UC et divers autres organes : des registres spécialisés (données, compteurs, adresses, états, etc), un séquenceur qui synchronise tous les composants, une horloge interne, une unité d'entrée-sortie qui gère la communication avec la mémoire (à ne pas confondre avec l'E/S des périphériques clavier, écran, etc). Le microprocesseur dispose selon son modèle d'un jeu (ensemble) d'instructions prédéfini.

S'il était tout seul, le microprocesseur ne pourrait pas faire grand chose. Au sein de l'architecture de Von Neumann seuls sont

représentés les composants logiques de base. Autour de ce schéma logique se raccordent bien d'autres organes électroniques comme les contrôleurs. Ces puces électroniques qu'on appelle aussi parfois chipsets sont aussi des sortes de microprocesseurs qui disposent souvent d'un jeu d'instructions pour les contrôler, justement. Ces instructions sont souvent moins nombreuses et pas généralistes. Les contrôleurs ont un rôle précis, selon leur genre : gérer un certain type de périphérique (ex : un contrôleur de carte graphique, un contrôleur pour les disques durs, etc), ou de transfert de données (ex : les contrôleurs des bus de mémoire et de données, les contrôleurs USB, PCI, etc). Tous ces composants sont intégrés sur un circuit imprimé principal appelé la carte mère.



Architecture de Von Neumann

Pour résumer : l'architecture de Von Neumann est simple à comprendre et répartit les fonctionnalités d'un ordinateur en quatre entités logiques. Ces entités logiques se retrouvent pour deux d'entre elles (UC et UAL) dans le microprocesseur. Les autres et les composants additionnels se retrouvent sur la carte mère ou sur les cartes d'extension (la mémoire n'est plus soudée sur une carte mère mais fournie sous forme de carte additionnelle appelée barrette, le contrôleur E/S graphique est sur une carte graphique additionnelle reliée à un bus PCI, AGP ou PCI/E). Les programmes sont exécutés par le microprocesseur qui est aidé (dans le sens où celui-ci accède aux fonctions proposées) par divers contrôleurs.

Les microprocesseurs actuels sont très complexes. Il n'est pas rare de trouver au sein de ceux-ci plusieurs UAL pour accélérer les traitements. De même, on trouve souvent une mémoire intermédiaire appelée mémoire cache ou antémémoire, celle-ci étant souvent spécialisée : une mémoire cache pour les instructions et une autre pour les données.

2. La machine de Turing

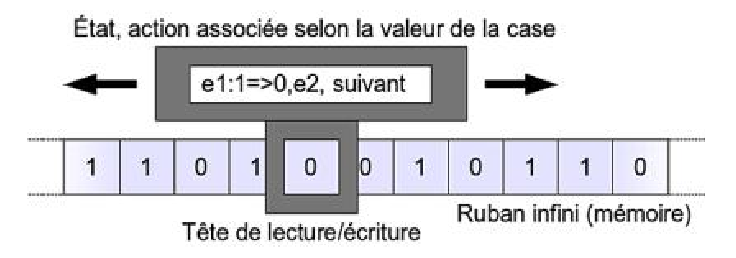
Avant même l'apparition des premiers vrais ordinateurs programmables, Alan Turing avait défini en 1936 (le 28 mai exactement) ce qu'on appelle la **Machine de Turing**. Cette machine abstraite (qui n'existe pas réellement) est en fait une méthode de modélisation du fonctionnement d'un ordinateur ou plutôt à l'origine d'un calculateur mécanique. Comment faire pour, depuis un postulat de base, arriver à un résultat donné ? En respectant des procédures données. C'est l'un des principes de l'algorithmique.

Une machine de Turing n'étant pas une vraie machine (au sens matériel), il suffit pour s'en servir soit de se servir de sa tête (réflexion et mémoire), soit d'un crayon qui fera office de **tête de lecture**, d'une longue bande de papier décomposée en cases qu'on appelle **ruban**, et d'une table de symboles et de procédures liée à l'**état** de la case à respecter quand on tombe sur une case contenant un symbole donné. On se place sur la première case, on vérifie son symbole et son état associés, on exécute la procédure associée (changement de valeur/symbole, avancer, reculer) et on continue à dérouler ce "programme" jusqu'à ce que la procédure vérifiant qu'on a obtenu le résultat final soit vérifiée. On vient de dérouler un programme, et l'ensemble symboles/procédure décrit ce programme. C'est l'ancêtre de l'algorithme.



Alan Turing, créateur de la machine abstraite du même nom

Il existe des livres complets sur la machine de Turing, notamment un de Alan Turing lui-même et de Jean-Yves Girard, aux Éditions Seuil, Collection Points Sciences. L'informatique n'est pas le seul domaine d'application de la machine. Elle permet de déterminer la complexité d'un algorithme, si quelque chose peut vraiment être calculé, a des domaines d'applications dans la physique et notamment l'optique, etc. Vous pouvez simuler une machine de Turing sur votre ordinateur via plusieurs langages dont un appelé Brainf*ck.



Exemple de machine de Turing

3. Représentation interne des instructions et des données

a. Le binaire

À quoi ressemblent les instructions et les données (valeurs) utilisées réellement par l'ordinateur ? Celui-ci ne comprend qu'une chose : des chiffres. Si l'être humain a inventé des représentations pratiques des chiffres avec le système décimal (soit une notation en base 10 en allant de zéro à neuf), un ordinateur ne manipule que deux valeurs : 0 ou 1. En effet si vous pouviez très fortement agrandir un circuit intégré, vous verriez que celui-ci est composé de nombreuses pistes dans lesquelles passe un courant électrique.

Dans ces circuits il n'y a que deux possibilités : soit le courant passe et dans ce cas cela équivaut à une valeur de un (1), soit le courant ne passe pas, et dans ce cas c'est la valeur zéro (0) qui est retenue. C'est du **binaire** (qui ne prend que deux valeurs). Une unité binaire s'appelle un **bit** (binary digit).Ce mot a été inventé par Claude Shannon en 1948.

Comme il y a plusieurs pistes sur les circuits, plusieurs valeurs 0 et 1, donc plusieurs bits, circulent en même temps. En associant ces valeurs, on obtient des valeurs plus grandes. En passant des données sur un fil, la valeur maximale est de 1. Si on prend deux fils, soit deux bits, la valeur maximale en binaire est 11, soit 3 en décimal. Pourquoi ? Voici une démonstration par étapes :

Courant Fil 1	Courant Fil 2	Binaire	Décimal
Ne passe pas	Ne passe pas	00	0
Ne passe pas	Passe	01	1
Passe	Ne passe pas	10	2
Passe	Passe	11	3

Une analogie fort ancienne (en informatique, ancien peut signifier un laps de temps très court), des années 1980, expliquait le fonctionnement des nombres binaires en associant des fils transportant du courant à des ampoules électriques. Chaque ampoule représente une valeur. Si le courant passe, l'ampoule s'allume et prend la valeur associée.

Le binaire, comme son nom l'indique, utilise une base deux (2) tout comme le décimal utilise une base dix (10). En **décimal**, tous les nombres peuvent être représentés à l'aide de puissances de 10. Prenez le nombre 1234 :

$1*103+2*102+3*101+4*10^0=1234$

L'unité est représentée par 10^0, la dizaine par 10^1, la centaine par 10^2, et ainsi de suite. Pour convertir le binaire en une valeur décimale plus lisible, il faut utiliser les puissances de 2, la plus petite valeur étant la plus à droite et est appelée **bit de poids faible**, la plus grande la plus à gauche et est appelée **bit de poids fort**. Dans l'exemple précédent, la valeur binaire 11 peut être convertie ainsi :

$1*2^1+1*2^0=2^1+2^0=2+1=3$

Les informaticiens et mathématiciens utilisent une notation particulière en indice (nombres en retrait bas) pour indiquer des conversions :

$11_{(2)}=3_{(10)}$

Voici un dernier exemple avec une valeur binaire codée sur 8 bits. Quelle est la plus grande valeur décimale qui peut être codée avec 8 bits ?

$2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0=128+64+32+16+8+4+2+1=255$

Donc

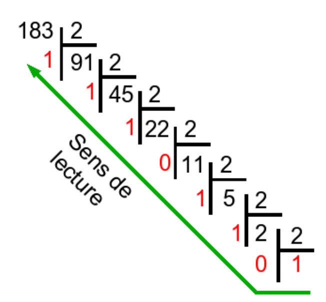
$11111111_{(2)}=255_{(10)}$

Soit 256 valeurs possibles de 0 (zéro) à 255, ce qui peut être plus simplement calculé par 2^n, soit deux puissance n, n étant le nombre de bits contenus dans la valeur binaire. La plus grande valeur convertie en décimal est donc :

2^n-1

Il est possible de convertir du décimal en binaire avec des divisions successives : on divise les résultats sans la virgule successivement par deux. Le résultat binaire sera la juxtaposition des restes (0 ou 1) sauf pour le dernier. Par exemple avec le nombre 183 :

- 183/2=91, reste 1
- 91/2=45, reste 1
- 45/2=22, reste 1
- 22/2=11, reste 0
- 11/2=5, reste 1
- 5/2=2, reste 1
- 2/2=1, reste 0
- On remonte du dernier : 10110111



Conversion décimale en binaire

C'est donc en binaire que sont représentées toutes les valeurs qu'un ordinateur manipule. C'est valable tant pour les données (numériques ou texte) que pour les instructions à destination du microprocesseur. Un nombre binaire correspondra à une instruction. Par exemple sur un microprocesseur x86 (Intel ou AMD), 01110100 est l'instruction *je* (jump if equal), ou encore la ligne 10110000 01100001 qui signifie *mov $0x61, %al* (placer la valeur hexadécimale 0x61 dans le registre AL).

Les ordinateurs du début des années 2000 savent manipuler des nombres sur 64 bits or 2^{64} est égal à 18 446 744 073 709 551 616 soit plus de 18 milliards de milliards !

L'idée d'utiliser deux valeurs pour encoder d'autres valeurs remonte à **Francis Bacon**. En 1623, il cherche une méthode stéganographique pour pouvoir crypter un texte composé des lettres de l'alphabet. Il remarque qu'un assemblage de deux lettres groupées par cinq permet de coder l'ensemble de l'alphabet. Il appelle cet alphabet "bilitère". Le "a" était représenté par "AAAAA", le "B" par "AAAAB", jusqu'au "Z", "BABBB". L'alphabet de l'époque, le latin, contenait vingt-quatre lettres, le "j" se confondant avec le "i" et le "u" avec le "v".

b. Les octets et les mots

Un ordinateur sait manipuler individuellement chaque bit d'une valeur. Mais les bits ne sont pas stockés individuellement dans une case mémoire. Ils sont regroupés, généralement par multiples de huit (8). Ainsi un ensemble de 8 bits est appelé un **octet**. L'avantage de l'octet est qu'il suffit (ou en tout cas a longtemps suffi) pour représenter tous les chiffres, lettres et symboles des alphabets occidentaux. Un octet représente les valeurs de 0 à 255.

Avec l'augmentation des espaces de stockages, de la quantité de mémoire, du besoin de représentation de nombres de plus en plus grands, d'un accès plus rapide à la mémoire ou encore de plus d'instructions, il a fallu augmenter la taille des valeurs à manipuler. De 8, puis 16, puis 32, certains microprocesseurs peuvent manipuler des valeurs de 64 voire 128 bits, parfois plus.. Ces valeurs deviennent difficiles à décrire et à représenter. Pour ces valeurs, on parle de **mot** mémoire (word en anglais). Certains microprocesseurs font une différence entre divers types de mots. Ceux de Motorola comme les 68000 (qui équipaient les ordinateurs Atari ST, Amiga, les consoles Sega Megadrive et plus récemment les Palm Pilot) utilisent des mots de 16 bits, et des mots longs (long word) de 32 bits.

Les instructions et les données sont donc codées sous forme de nombres binaires qu'on appelle des mots. Cependant suivant le type de microprocesseur l'ordre des mots est différent entre la réalité et son stockage en mémoire. Avec un microprocesseur x86 en mode réel (16 bits) le nombre décimal 38457 nécessite 16 bits soit deux octets ou un mot de seize octets pour être représenté :

$38457_{(10)}=1001011000111001_{(2)}$

Pour stocker cette valeur en mémoire, les 8 premiers bits de poids faible, soit l'octet de poids faible, seront placés dans la première case mémoire, et les 8 derniers bits de poids fort, soit l'octet de poids fort, seront placés dans la case suivante. La démarche serait la même en 32 bits ou 64 bits.

Case mémoire 1	Case mémoire 2
00111001	10010110

c. L’hexadécimal

Si vous reprenez l’exemple d’une valeur binaire codée sur 64 bits, il faut 64 0 ou 1 pour la décrire :

1111111111111111111111111111111111111111111111111111111111111111

En décimal il faut vingt chiffres : 18 446 744 073 709 551 616

Cela prend beaucoup de place et c’est difficile à manipuler. Puisqu’il existe une base 10 (décimal) et une base 2 (binaire), pourquoi ne pas prendre une base plus élevée, multiple de 2, pour réduire la longueur et la manipulation de ces nombres ? C’est ainsi qu’en informatique il est d’usage d’utiliser la base 16, appelée hexadécimale.

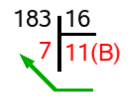
Une base hexadécimale permet de coder les valeurs de 0 à 15. Si de 0 à 9 on utilise les valeurs décimales correspondantes, au-dessus il faut utiliser des lettres de l’alphabet, de A (10) à F (15).

Comment passer du binaire à l’hexadécimal ? Ceci revient à répondre à la question "Combien faut-il de bits dans un nombre binaire pour coder 16 valeurs ?" $2^4=16$. Il faut donc 4 bits. Le tableau suivant résume les conversions.

Décimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexa	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binaire	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Si vous reprenez le nombre 183 qui nécessite 8 bits soit un octet, sa conversion donne B7 en hexadécimal. On dit donc que :

$183_{(10)}=B7_{(16)}$



du décimal à l’hexadécimal

Si vous prenez la valeur maximale en 64 bits, cela donne FFFFFFFFFFFFFFFF soit 16 caractères. Un informaticien exercé est quasiment capable de convertir à la volée de l’hexadécimal en décimal. Prenez la valeur 8C soit 10001100 en binaire. Le C vaut 12.

8*16+12=140

Sans aller plus loin, sachez que les bases 2, 10 et 16 ne sont pas les seules. Sur certaines machines et certains systèmes d’exploitation, il est courant d’utiliser une base 8 pour des valeurs ne nécessitant que trois bits pour être représentée. Somme toute, tant qu’il reste assez de symboles, rien n’empêcherait d’avoir une base 30 !

L'algorithmique

1. Programmer, c'est un art

Pour obtenir un résultat donné, il faut généralement suivre une méthode, une certaine logique. Sauf à être un grand pâtissier dont la science des mélanges des ingrédients est innée (ou le fruit d'une longue pratique), vous n'obtiendrez jamais un délicieux gâteau au chocolat même si vous disposez des meilleurs ingrédients et accessoires de cuisson, si vous ne connaissez pas les bonnes proportions, l'ordre dans lesquels ajouter les ingrédients, le temps de cuisson, la température : bref, la recette. De même, sans formation de mécanicien ou sans la documentation technique du moteur de votre véhicule, inutile de vous lancer dans un changement de joint de culasse : c'est la casse assurée.

Il en est de même de la programmation. Il existe plusieurs langages de programmation très simples, extrêmement simples parfois, qui peuvent donner un temps l'illusion que vous savez programmer. En entreprise même, certains employés sont bombardés développeurs pour leurs quelques connaissances confuses de Visual Basic, de Delphi ou de Windev. Le résultat risque d'être catastrophique. Les publicités sont alléchantes mais trompeuses. Les bons programmeurs, y compris les autodidactes, ont tous à un moment ou un autre eu affaire avec les algorithmes, car il existe en programmation une multitude de moyens d'arriver à un résultat, mais très peu pour obtenir le meilleur résultat possible, ce qui explique pourquoi beaucoup de programmes ayant la même fonction, se ressemblent (au niveau de la programmation) alors que ce ne sont pas les mêmes programmeurs qui les ont développés. Les débutants qui se lancent dans des projets de programmation audacieux se retrouvent parfois bloqués, ne maîtrisant pas une technique particulière de logique de programmation. Certains abandonnent, d'autres trouvent un moyen de contournement (souvent peu reluisant). Les derniers liront peut-être un livre d'algorithmique comme celui-ci, qui à défaut de donner une solution complète à leur problème, leur fournira les bases et les techniques pour avancer.

Les ordinateurs personnels du début des années 1980 étaient tous livrés soit avec un langage BASIC inclus directement dans la machine (en ROM), soit sur une cartouche, cassette ou disquette annexe. Le Basic de Microsoft (Qbasic, Quickbasic) était livré avec le DOS du PC. Les Amstrad avaient le basic Locomotive, les Atari ST l'Atari Basic et surtout le GFA Basic, un langage de grande classe, etc. Une génération complète d'utilisateurs s'est lancée dans la programmation à l'aide de ces langages et de la documentation fournie qui bien souvent fournissait non seulement les références du langage mais aussi les méthodes de base de programmation. Avec plus ou moins de succès. Le résultat était souvent un infâme bidouillage, mais qui marchait.

Or le but n'est pas que le programme fonctionne, mais qu'il fonctionne vite et bien, bref le mieux possible. Le meilleur ordinateur au monde et le meilleur langage au monde ne vous y aideront pas.

2. Définition : l'algorithme est une recette

Avez-vous déjà eu l'occasion de programmer un magnétoscope (en voie de disparition) ou un enregistreur de dvd ? Qu'avez-vous fait la première fois que vous avez allumé votre poste de télévision pour régler la réception des chaînes ? Nul doute que vous avez ouvert le mode d'emploi et suivi la séquence d'instructions indiquée : appuyer sur la touche **Menu** de la télécommande, se déplacer sur **Enregistrement** et appuyer sur **OK**, se déplacer sur une ligne puis indiquer la chaîne, l'heure, etc.

Avez-vous déjà eu l'occasion de faire la cuisine ? Pour un gâteau, vous êtes-vous lancé directement ou avez-vous ouvert un livre pour récupérer la liste et la quantité de chaque ingrédient, pour suivre la recette : faites fondre le chocolat et le beurre dans une casserole à feu doux, retirez la casserole du feu, incorporez les jaunes d'œuf, puis le sucre et la farine, battez les œufs en neige puis incorporez doucement dans le mélange, etc.

Dans les deux cas, félicitations ! Vous avez déroulé votre premier algorithme !

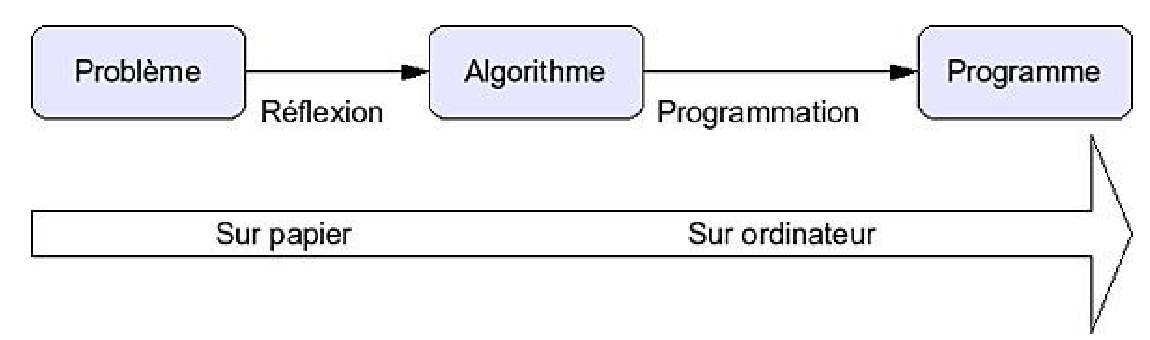
Une définition simple d'un algorithme : c'est une suite d'instructions qui, quand elles sont exécutées correctement aboutissent au résultat attendu. C'est un énoncé dans un langage clair, bien défini et ordonné qui permet de résoudre un problème, le plus souvent par calcul. Cette définition est à rapprocher du fonctionnement de la machine de Turing qui avant l'apparition de l'ordinateur utilisait cette démarche pour résoudre de nombreux problèmes. L'algorithme est donc une recette pour qu'un ordinateur puisse donner un résultat donné.

Le mot algorithme vient du nom du mathématicien Al Khuwarizmi (Muhammad ibn Mūsā al-Khuwārizmī), savant persan du IXème siècle, auteur d'un ouvrage appelé "La transposition et la réduction", Al-jabr wa'l-muqābalah. Le mot Al-jabr deviendra algèbre, le nom de l'auteur sera latinisé en Algoritmi, qui sera à la base du mot algorithme.

3. Pourquoi utiliser un algorithme ?

L'algorithme décrit formellement ce que doit faire l'ordinateur pour arriver à un but bien précis. Ce sont les instructions qu'on doit lui donner. Ces instructions sont souvent décrites dans un langage clair et compréhensible par l'être humain : faire ceci, faire cela si le résultat a telle valeur, et ainsi de suite.

Un algorithme bien établi et qui fonctionne (tout au moins en théorie) pourra être directement réécrit dans un langage de programmation évolué comme le C, Java ou PHP. Malheureusement, en programmation c'est souvent à l'homme de se mettre au niveau de la machine.



De la réflexion à la programmation

Plus que cela, un algorithme décrit une méthode de résolution de problèmes courants. Un algorithme est donc réutilisable, sauf cas ponctuel ou très précis. Il existe plusieurs moyens d'obtenir un même résultat, mais certains sont meilleurs que d'autres. C'est le cas par exemple des méthodes de tris de données par ordre alphabétique. Il existe divers algorithmes décrivant ces méthodes, certaines étant adaptées à des quantités plus ou moins importantes de données.

La maîtrise de l'algorithmique et l'apprentissage des algorithmes de base sont une des conditions de la réussite d'un projet en programmation, qu'il soit personnel ou professionnel. L'expérience aidant, vous allez acquérir au fur et à mesure des mécanismes de pensée qui vous permettront d'optimiser les traitements que vous devez programmer, tant en vitesse qu'en occupation mémoire ou même en quantité de lignes de programmation. Sur ce dernier point, il existe de nombreux cas où des algorithmes longs et complexes sont plus performants que d'autres semblant plus pratiques au premier abord.

Apprendre l'algorithmique (ou l'algorithmie, les deux sont autorisés) c'est donc apprendre à programmer dans les règles de l'art. Tout au long de cet ouvrage, vous allez découvrir les notions élémentaires qui vous permettront tant de comprendre le fonctionnement interne d'un programme que de le concevoir, à l'aide d'une progression simple et constante et d'exemples pratiques et compréhensibles.

4. Le formalisme

Le but d'un algorithme étant de décrire un traitement informatique dans quelque chose de compréhensible par l'humain (et facilement transposable vers la machine), pour qu'un algorithme soit compréhensible, il faut qu'il soit clair et lisible. Dans ce cas, il existe deux moyens efficaces :

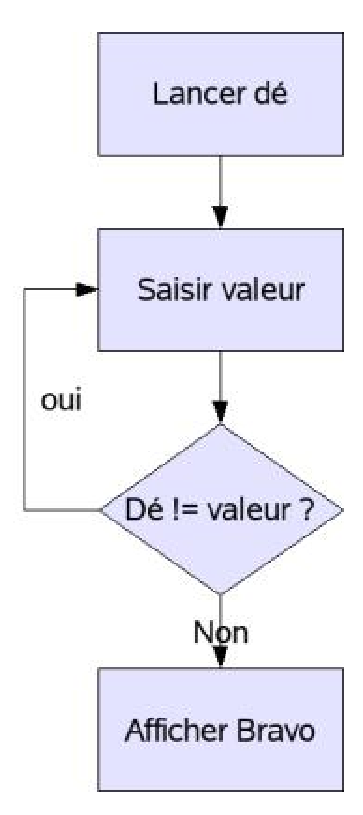
- soit d'écrire l'algorithme sous forme de texte simple et évident (faire ceci, faire cela),
- soit de faire un schéma explicatif avec des symboles.

Dans la pratique, les deux formes sont possibles. Mais un dessin ne vaut-il pas un long discours ? Il est d'ailleurs courant de commencer par un schéma, puis quand celui-ci devient trop complexe, de passer à un texte explicatif (la recette).

Dans les deux cas, la syntaxe pour le texte ou les symboles pour les schémas doivent répondre à des règles strictes, voire normalisées. Il faut que chacun connaisse leur signification et sache donc les interpréter. C'est pour cela que toutes les représentations algorithmiques suivent à peu de choses près le même formalisme. Si les schémas sont possibles, ils sont cependant moins utilisés que les algorithmes sous forme textuelle. C'est que si vous construisez un algorithme, il est plus facile de le corriger quand il est saisi au clavier sous forme de texte que lorsqu'il est dessiné sous forme d'organigramme dans un logiciel de dessin vectoriel ou de présentation.

a. La représentation graphique

Les algorithmes peuvent être construits à l'aide de symboles d'organigrammes. Les étudiants en informatique (BTS, DUT) connaissent bien cette tablette en plastique permettant de dessiner des organigrammes. Ils l'utilisent en algorithmique, en base de données, en méthode Merise, etc (dans chaque cas la signification est différente). Voici un exemple d'algorithme sous forme d'organigramme qui simule un lancé de dé et qui demande à une personne de deviner la valeur.



Un formalisme qui occupe trop de place

Dans cet exemple simplifié, les traitements sont dans des rectangles, les prises de décision dans des losanges, et les flèches représentent l'ordre du déroulement du programme. Si une valeur est présente à côté de la flèche, l'action dépend du résultat de la question posée dans le losange. Les décisions et les flèches peuvent décrire des boucles. Dans le schéma, tant que l'utilisateur n'a pas saisi la bonne valeur, la question lui est de nouveau posée.

Cet algorithme est très simple, l'organigramme aussi. Cependant voyez déjà la taille de celui-ci (la place qu'il prend) par rapport à ce qu'il fait. Imaginez maintenant un algorithme plus complexe qui doit par exemple décrire tous les cas de figure dans la gestion d'une communication entre deux machines (description d'un protocole de communication) : le schéma nécessitera une feuille d'une grande dimension et sera difficile à étudier.

b. L'algorithme sous forme de texte

Prenez le même énoncé du lancé de dé. Celui-ci pourrait être écrit ainsi en français correct :

- 1ère étape : lancer le dé.
- 2ème étape : saisir une valeur.
- 3ème étape : si la valeur saisie est différente de la valeur du dé, retourner à la troisième étape, sinon continuer.
- 4ème étape : afficher "bravo".

Vu ainsi, c'est très simple. De cette manière, il est évident que tout le monde, même un non-informaticien, comprend ce que l'algorithme est censé faire. Cependant, si les algorithmes complexes devaient être écrits ainsi, ce serait encore une fois bien trop long et vous finiriez soit par vous lasser d'une écriture trop complexe, soit cela prendrait trop de place. C'est pourquoi il faut utiliser une syntaxe précise et concise.

```
/* Commentaires : ce programme affiche bonjour */
PROGRAMME HelloWorld

/* Déclarations : variables, constantes, types, etc */
VAR
    de:entier,
    valeur:entier
```

```
/* Début du programme */
DEBUT
      de←aléatoire(6)
      valeur←0
      Tant que valeurde Faire
             Lire valeur
      FinTantQue
      Afficher "Bravo"
FIN
```

Si vous comprenez déjà le programme ci-dessus alors cet ouvrage vous sera encore plus agréable à lire. Sinon, la suite vous donnera de toute façon toutes les explications nécessaires à la compréhension de chaque ligne de cet algorithme. Il reprend de manière très détaillée toutes les étapes à suivre. Sous cette forme, il est presque possible d'implémenter l'algorithme ligne à ligne dans un langage de programmation évolué.

C'est sous cette forme textuelle que les algorithmes seront représentés dans ce livre. Ce texte, programme ou pseudo-code algorithmique, est décomposé en plusieurs parties :

- Le nom du programme, qui n'amène pas de commentaires particuliers, situé après le mot "**PROGRAMME**".
- Une zone de déclaration des données utilisées par le programmes : variables, constantes, types, structures, tableaux, etc. Si la signification de ces mots vous échappe, ceux-ci seront expliqués au fur et à mesure des différents chapitres. Cette zone commence par le mot "**VAR**".
- Le programme lui-même, c'est-à-dire les divers traitements. Les instructions du programme sont encadrées par les mots "**DEBUT**" et "**FIN**". Il vous est conseillé, pour plus de clarté et de lisibilité, d'indenter les diverses lignes (de les décaler les unes par rapport aux autres) à l'aide des touches de tabulation. Le programme peut être de n'importe quelle longueur : une ligne ou 10000 lignes, ceci n'a pas d'importance.
- Les commentaires : c'est un texte libre qui peut être étendu sur plusieurs lignes et encadré par les séquences de caractères "/*" et "*/". Si votre commentaire tient sur une seule ligne, vous pouvez uniquement la commencer par les caractères "**//**".
- Une dernière partie, ou plutôt première car lorsqu'elle est présente elle se situe avant toutes les autres, peut être constituée des sous-programmes, semblants de programmes complets appelés par le programme principal. Ces sous-programmes, appelés procédures ou fonctions, font l'objet d'un chapitre complet.

5. La complexité

L'exemple du lancé de dé est un algorithme très simple, court, concis et rapide. Ce n'est pas le cas de tous les algorithmes. Certains sont complexes et le traitement résultant peut nécessiter beaucoup de temps et de ressources de la machine. C'est ce qu'on appelle le "coût" de l'algorithme, et il est calculable. Si un algorithme est "gourmand" son coût sera plus élevé. Il existe certains cas où il est possible d'utiliser plusieurs algorithmes pour effectuer une même tâche, comme pour trier les éléments d'un tableau de valeurs. Certains algorithmes se révèlent être plus coûteux que d'autres, passé un certain nombre d'éléments à trier. Le coût d'un algorithme reflète sa complexité ou en terme plus simple son efficacité. Les mots "coût", "complexité" et "efficacité" reflètent ici la même définition. Plus un algorithme est complexe, plus il est coûteux et moins il est efficace. Le calcul de cette complexité a comme résultat une équation mathématique qu'on réduit généralement ensuite à une notion d'ordre général.

La complexité est notée O(f(n)) où le O (grand O) veut dire "d'ordre" et f est la fonction mathématique de n qui est la quantité d'informations manipulée dans l'algorithme. Voici un exemple pour mieux comprendre : soit un algorithme qui compte de 1 à n et qui affiche les valeurs correspondantes. Dans la pratique, vous allez utiliser une boucle (voir le chapitre Les boucles) allant de 1 à n. Il faudra faire n passages pour tout afficher et donc vous aller manipuler n fois l'information. La fonction mathématique donnant le coût sera alors f(n)=n. La complexité est alors linéaire et vous la noterez O(n).

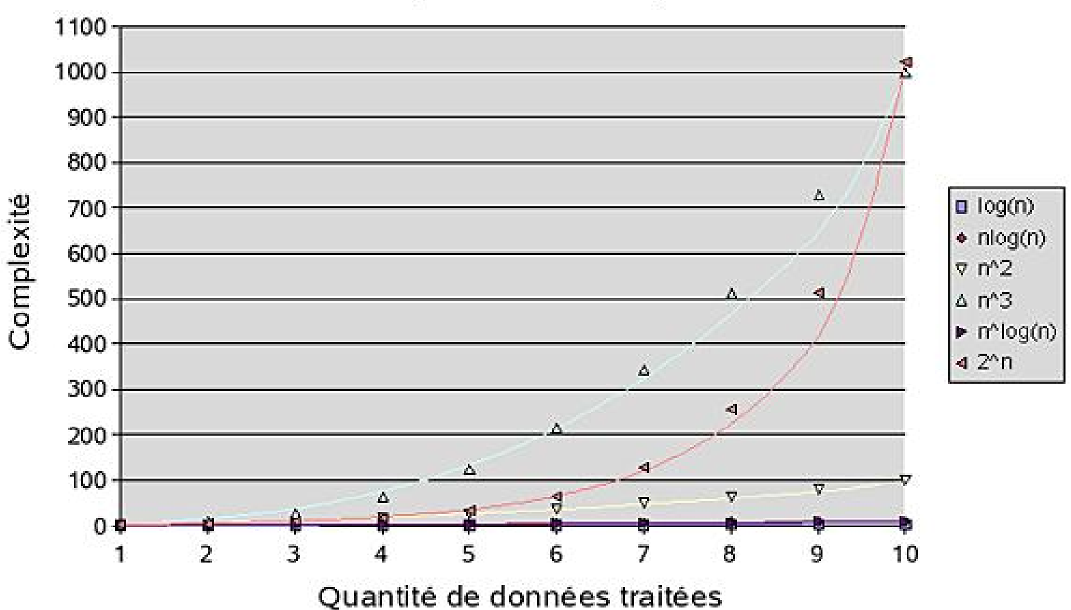
Si dans le même algorithme vous décidez de faire une seconde boucle dans la première, pour afficher par exemple une table de multiplications : la première boucle va toujours de 1 à n, la seconde va aussi de 1 à n. Au total vous obtenez n fois n boucles, donc n^2 boucles. La complexité est donc $f(n)=n^2$, et vous la noterez $O(n^2)$. Le coût de l'algorithme augmente au carré du nombre d'informations.

Si vous ajoutez en plus une quelconque opération dans la première boucle, cette opération a aussi un coût que vous pouvez tenter de prendre en compte. Si vous ajoutez une multiplication et que celle-ci a un coût de 1, alors la complexité finale est de n*(n+1) soit n^2+n. Cependant si vous faites une courbe pour de grandes valeurs de n et que vous comparez avec la courbe simple n^2, vous remarquerez que le rajout devient négligeable. Au final, l'algorithme conserve une complexité $O(n^2)$.

Si la complexité peut parfois être calculée assez finement, il en existe plusieurs "prédéfinies" :

- O(1) : complexité constante
- O(log(n)) : complexité logarithmique
- O(n) : complexité linéaire
- O(n.log(n)) : complexité quasi-linéaire
- O(n^2) : complexité quadratique
- O(n^3) : complexité cubique
- O(n^p) : complexité polynomiale
- O($n^{log(n)}$) : complexité quasi-polynomiale
- O(2^n) : complexité exponentielle
- O(n!) : complexité factorielle

Ces complexités ne sont pas forcément faciles à appréhender, aussi voici un graphique représentant quelques-unes de celles-ci. En abscisse est indiqué le nombre de données à traiter et en ordonnée la complexité associée : le nombre d'opérations effectuées pour n données. Pour des complexités d'ordre O(2^n) l'algorithme effectue déjà 1024 opérations et plus de 3,5 millions pour O(n!) !



Courbes de complexité

Comment se représenter réellement une complexité en terme de temps passé par l'ordinateur à traiter les données ? Chaque microprocesseur est capable de traiter un certain nombre d'opérations par seconde. Le plus long étant généralement les calculs sur les réels (flottants), le critère souvent retenu pour déterminer la puissance brute d'un processeur est le **FLOPS** : Floating Point Operations Per Second. Un Intel Pentium 4 à 3.2 GHz tourne à une moyenne de 3,1 GFLOPS (GigaFlops) soit 10^9 FLOPS, ou encore un milliard d'opérations sur réels par seconde. Si vous traitez 20 données dans un algorithme de complexité O(n), la vitesse de calcul se chiffre en millionièmes de seconde. Le même nombre de données dans un algorithme de complexité O(n!) doit effectuer 2432902008176640000 opérations ce qui prendra 784807099 secondes, ou encore une fois converti autour de 25 ans ! Bien entendu, une complexité O (n!) est la pire qui puisse exister. Avec une complexité inférieure O(2^n), le traitement prendrait un dixième de seconde tout de même, ce qui est énorme et relativise fortement la puissance des processeurs…

Vous comprenez maintenant l'utilité de connaître la complexité des algorithmes et d'optimiser ceux-ci…

Dans la suite, les complexités ne seront fournies que dans les cas où les traitements, plus compliqués que d'habitude, sont en concurrence avec diverses méthodes. C'est le cas par exemple des méthodes de tris sur des tableaux. Ceci dans l'unique but de vous donner un simple ordre d'idée.

Les langages d'implémentation

1. Quel langage ?

Il existe plusieurs centaines de langages de programmation si on tient compte de toutes les variantes possibles d'un même langage. Comme vous avez pu le lire au début de ce chapitre, l'ordinateur ne comprend nativement qu'un seul langage, le langage machine. Croyez-vous vraiment que vous allez implémenter le programme de lancer de dé directement en binaire (ou même en hexadécimal) ? Le choix du langage mérite une petite démonstration. On a coutume dans le milieu de l'informatique, de tester un langage en lui faisant afficher un message pour dire bonjour, en l'occurrence le fameux "Hello world!". Voici comment afficher ce texte dans divers langages :

En assembleur x86 sous DOS

```
Cseg segment
assume cs:cseg, ds:cseg
org 100h
main proc
jmp debut
mess db 'Hello world!$'
debut:
mov dx, offset mess
mov ah, 9
int 21h
ret
main endp
cseg ends
end main
```

En shell Unix

```
echo "Hello world!"
```

En Basic originel

```
10 PRINT "Hello world!"
20 END
```

En COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-WORLD.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
DISPLAY "Hello world!".
STOP RUN.
```

En langage C

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello world!\n");
    return 0;
}
```

En langage C++

```
#include <iostream>
```

```
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

En PHP

```
<?php
 print ("Hello world!");
?>
```

En Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

En Visual Basic

```
Sub Main()
  MsgBox("Hello world!")
End Sub
```

En Pascal

```
program Bonjour;
begin
  WriteLn(‘Hello world!’);
end.
```

2. Classifications des langages

Que remarquez-vous ? Il y a autant de syntaxes différentes qu’il existe de langages. Cependant vous constatez que les langages C, C++, Java ou PHP ont de nombreuses ressemblances, alors que l’assembleur ou le COBOL semblent sortis d’ouvrages de Science-Fiction. C’est que les premiers ont quelques liens familiaux tandis que les autres sont radicalement opposés.

a. Haut niveau, bas niveau

Puisqu’il existe des centaines de langages de programmation, lequel choisir pour implémenter vos algorithmes ? Il n’y a pas de réponse simple à cette question. Chaque langage a été généralement conçu pour des usages différents et souvent spécifiques, bien qu’en évoluant la plupart des langages dits de haut niveau soient devenus de plus en plus généralistes. Il existe plusieurs classifications des langages. La plus ancienne dépend de l’affinité du langage par rapport à la machine. On parle alors du niveau du langage. Il est courant de parler d’un langage de bas niveau ou de niveau zéro (0) quand celui-ci nécessite des connaissances approfondies du fonctionnement de votre ordinateur : mécanismes de gestion de la mémoire, instructions du microprocesseur, etc. Un exemple de langage de très bas niveau est le langage machine sous sa forme binaire, ou de le la programmation en assembleur où ces mêmes valeurs binaires sont représentées par des mots (mnémoniques) en anglais. Vu que les programmes sont directement compréhensibles par le matériel, vos programmes seraient alors les plus rapides. Il est tout à fait possible de programmer de grosses applications en assembleur (et notamment des jeux), c’était d’ailleurs très courant jusqu’à l’apparition des machines très rapides où leur vitesse a compensé une plus faible vitesse d’exécution d’un langage plus évolué comme le C, mais avec l’avantage d’une programmation plus simple.

À l’opposé des langages de bas niveau se trouvent les langages de haut niveau. Il n’y a pas d’échelle précise. Vous pourrez trouver dans quelques sources des niveaux allant de 0 à 4, mais les langages évoluant tellement vite, certains langages qui étaient considérés de haut niveau comme le C se sont vus déclassés vers le bas ! Un langage de haut niveau permet de faire une abstraction presque complète du fonctionnement interne de votre ordinateur. Le langage (ou plutôt son compilateur ou son interpréteur) se chargera de convertir vos ordres simples (en apparence) en langage de bas niveau (en langage machine). Ne vous fiez pas aux apparences : l’affichage d’une

boîte de dialogue prend une ligne de langage de haut niveau, mais des centaines en assembleur ! Parmi les langages de très haut niveau se trouvent Java, C#, le PHP, ou même le C (en faisant abstraction de certains mécanismes). L'inconvénient d'un langage de haut niveau est qu'il n'est pas toujours possible d'aller dans les détails les plus fins.

b. Diverses classifications

À côté des niveaux, tous les langages n'ont pas le même but. Il n'est pas possible de comparer le langage HTML dont le rôle est de formater des pages web et le Visual Basic qui permet un développement rapide d'applications graphiques. C'est pourquoi il existe d'autres classifications dont voici un bref échantillon :

- généraliste ou spécialisé
- objet ou procédural
- typé ou non typé (cf. chapitre Les variables et opérateurs)
- interprété ou compilé
- etc

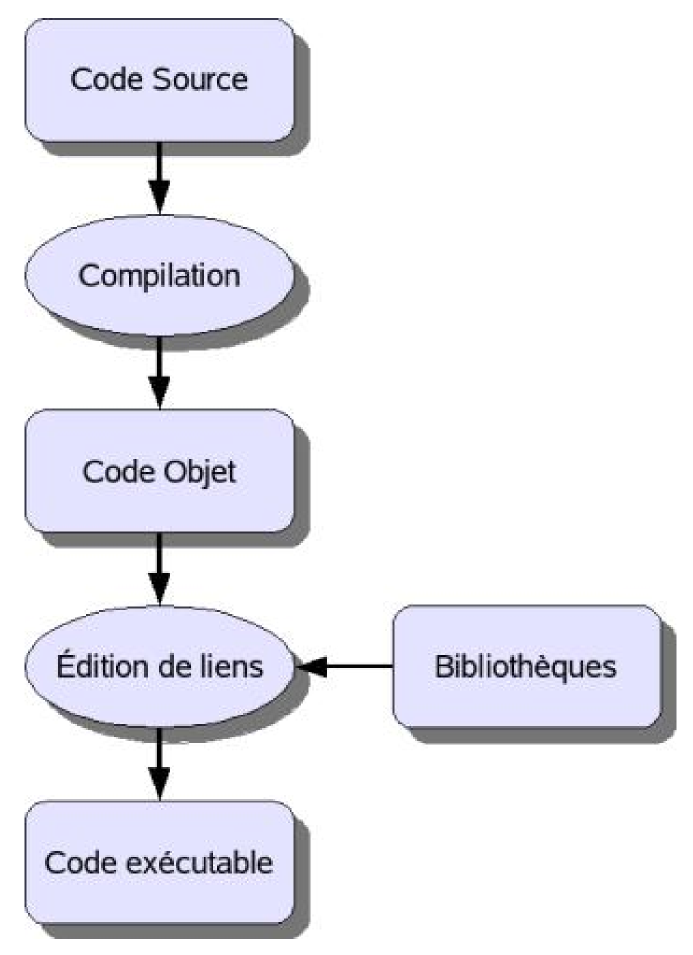
Certains langages sont spécialisés. Le HTML est spécialisé dans la conception de pages web statiques : son exécution a comme résultat direct l'affichage d'une page HTML qu'il a mis en forme. Le SQL est un langage de base de données : il permet de gérer des enregistrements de données. Le Javascript est un langage qui permet de programmer des pages web dynamiques du côté du navigateur web, tandis que le PHP (bien que devenu généraliste) ou ASP permettent de programmer des sites web dynamiques mais cette fois du côté du serveur. Certains langages peuvent faire appel à d'autres langages. Vous pouvez parfaitement faire du SQL dans du PHP, si votre site doit accéder à une base de données...

c. Compilé ou interprété

Une autre distinction à prendre en compte est la différence entre un langage interprété et un langage compilé. Un langage est dit compilé quand le programme source sous forme de texte est tout d'abord lu et traité par un autre programme appelé compilateur qui le convertit en langage machine directement compréhensible par l'ordinateur. Vous tapez votre programme, vous lancez la commande de compilation et enfin vous obtenez un fichier exécutable (un .exe sous Windows par exemple) que vous pouvez le cas échéant lancer comme n'importe quel autre programme en langage machine.

Un programme en langage interprété nécessite pour fonctionner un interprète (ou interpréteur) qui est un autre programme qui va traduire directement, au fur et à mesure de son exécution, votre programme en langage machine, un peu comme un vrai interprète qui dans une interview traduit simultanément l'anglais en français. Le programme est souvent un fichier texte et l'interprète analyse la syntaxe de celui-ci avant de le dérouler dynamiquement.

Un programme interprété sera plus lent qu'un langage compilé à cause de la conversion dynamique du programme, alors que cette étape est déjà effectuée à l'avance avec un langage compilé. Au contraire, la correction des erreurs est plus simple avec un langage interprété. Comme la correction et le résultat sont immédiats, l'interprète va vite vous indiquer au cours de l'exécution où se trouve l'erreur de syntaxe (mais pas de logique) lorsqu'il va la rencontrer, à quelle ligne, l'instruction en cause, éventuellement une aide supplémentaire. Alors qu'avec un compilateur, c'est au moment de la compilation, souvent longue, qu'apparaissent les erreurs. Une fois compilé, d'autres erreurs plus complexes comme les fuites mémoire peuvent apparaître mais il devient difficile d'en déterminer l'origine (il faut alors faire appel à d'autres programmes spéciaux appelés débuggers).

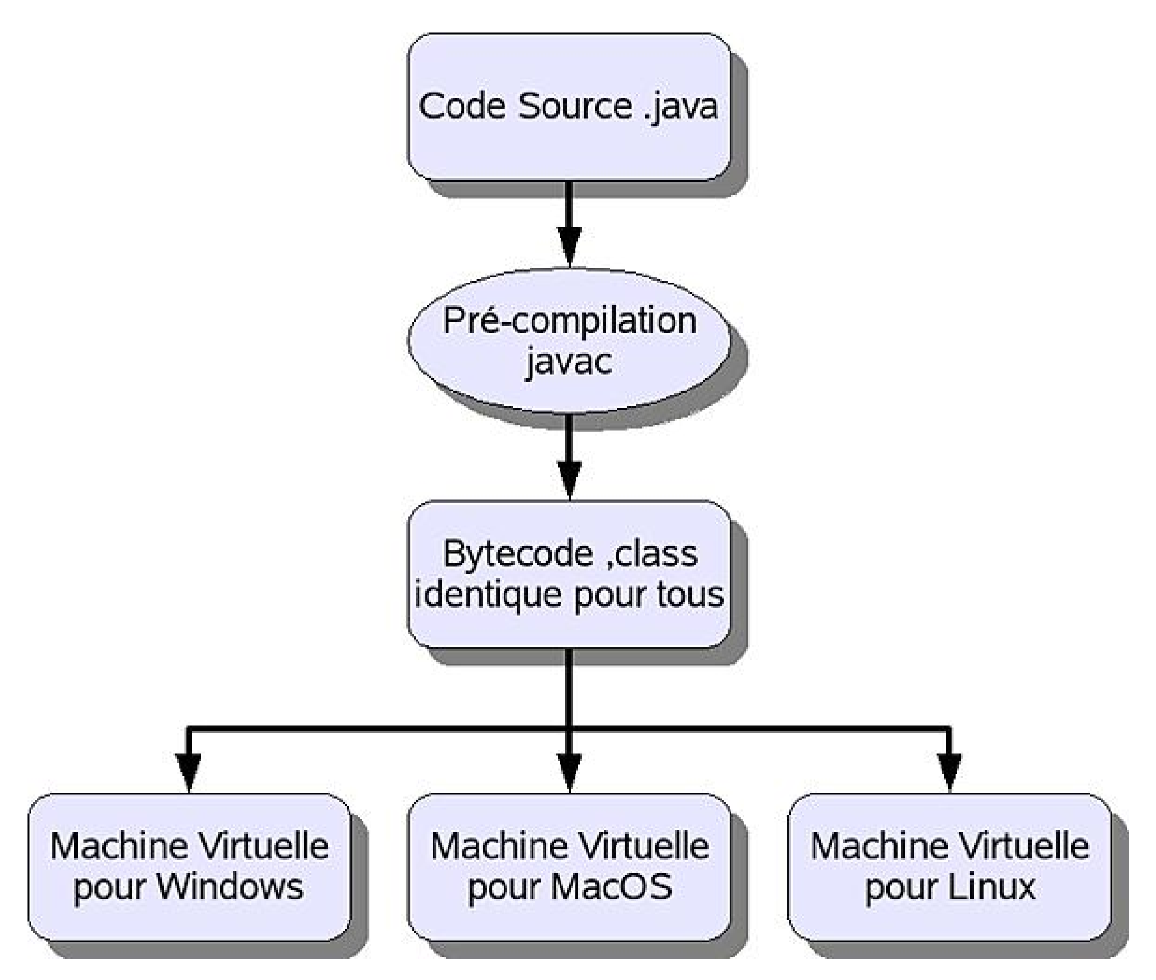


Étapes de compilation et d'édition des liens en C

3. La machine virtuelle

Il existe une étape intermédiaire entre l'interprété et le compilé : la machine virtuelle applicative. La machine virtuelle est un programme, généralement un interpréteur, qui permet d'isoler l'application qu'il doit faire tourner du matériel et même du système d'exploitation. Le programme n'a théoriquement aucun accès aux spécificités du matériel, l'ensemble de ses besoins lui étant fourni par la machine virtuelle. Ainsi, tout programme conçu pour cette machine virtuelle pourra fonctionner sur n'importe quel ordinateur, du moment que ladite machine virtuelle existe pour cet ordinateur. C'est en quelque sorte une couche d'abstraction ultime. Généralement, le programme fonctionnant depuis la machine virtuelle a déjà subi une première phase de compilation pour le transformer non pas en langage machine propre à l'ordinateur, mais dans un langage "machine virtuelle" pour ainsi dire, que l'on nomme **bytecode**. Ce bytecode pourra être interprété par la machine virtuelle, ou plutôt et ceci de plus en plus régulièrement, compilé à la volée juste au moment de son utilisation (technologie JIT, *Just in Time*).

Ainsi dans certaines circonstances le programme fonctionne presque aussi vite qu'un programme compilé pour une machine cible ! Un exemple de langage utilisant une machine virtuelle est Java.



Génération et exécution de bytecode Java

Pour implémenter vos algorithmes, il vous faut trouver un langage simple, de haut niveau, généraliste mais vous permettant d'évoluer vers des applications complexes et complètes. Il ne faut pas que vous ayez à passer par des étapes longues et complexes de compilation et il faut que le langage propose une ouverture vers d'autres langages. Dans un esprit d'ouverture et de compatibilité, il serait intéressant que ce langage ne soit pas disponible uniquement sous Windows et que si possible le programme résultant puisse fonctionner sur plusieurs systèmes d'exploitation sans avoir à le modifier, ni à le recompiler. Plusieurs langages pourraient convenir. Le mieux est de choisir un langage interprété : le PHP.

4. PHP

a. Les avantages

PHP, (*PHP Hypertext Preprocessor*), dispose de toutes les qualités nécessaires. Il est disponible sur la plupart des architectures et systèmes d'exploitation. PHP est à l'origine un langage de script HTML exécuté du côté du serveur, mais c'est aussi un langage complet disposant de nombreuses fonctionnalités lui permettant de faire fonctionner des programmes depuis la ligne de commande ou depuis un bureau. Il est notamment possible de concevoir des applications graphiques complètes en PHP, directement, sans passer par du HTML. Sa syntaxe est largement inspirée du langage C, de Java et de Perl, avec des améliorations spécifiques. Le but initial du langage était d'écrire rapidement des pages HTML dynamiques.

b. Historique

L'origine de PHP remonte à 1995 quand Rasmus Lerdorf a créé PHP/FI, une librairie de scripts Perl utilisée pour la publication de son CV sur son site personnel. Au fur et à mesure des évolutions, la librairie a été portée en C et agrémentée de nouvelles fonctionnalités pour créer des pages dynamiques simples pour le web et accéder à quelques sources de données. PHP/FI signifie Personal Home Page/Forms Interpreter.

PHP/FI 2.0 sort en 1997, toujours développé par une seule personne. C'est le succès : 50000 sites (1% des noms de domaines) disent l'utiliser ou le supporter. Les contributeurs affluent.

PHP 3.0 sort en juin 1998, c'est la première version développée conjointement par Rasmus Lerdorf, Andi Gutmans et Zeev Suraski et entièrement réécrite (les versions précédentes étaient trop lentes pour une application commerciale). Outre ses performances, PHP 3 est modulaire et extensible ; il est possible de lui greffer des API supplémentaires, ce qui n'a pas manqué d'être fait avec le support de nombreuses bases de données, de formats et de protocoles. Il dispose d'une syntaxe plus cohérente et d'un support basique de l'objet. Il a été installé sur 10% du parc mondial des serveurs web avec des dizaines de milliers de développeurs et des centaines de milliers de sites.

Le moteur de PHP 4 a été repensé afin d'en améliorer les performances pour des applications complexes et d'augmenter encore sa modularité. Un nouveau moteur a vu le jour, appelé Zend Engine (contraction de Zeev et Andi). PHP 4 sort officiellement en mai 2000 et apporte des performances accrues, le support de plusieurs serveurs web, les sessions, une sécurité accrue.

PHP 5, sorti en juillet 2004, propose, entre autres, une amélioration des performances du moteur Zend (Zend Engine 2), un modèle objet étendu et très proche du C++, une compatibilité accrue avec de nouveaux standards et les anciens (RPC, XML, .NET...). C'est actuellement la dernière version stable.

c. Informations pratiques

PHP peut être utilisé pour n'importe quel projet, dont :

- les forums et messageries
- le commerce électronique
- la banque, les comptes en ligne, les paiements en ligne
- la publication en ligne
- les moteurs de recherche

Tout ce que vous voulez, en fin de compte !

Voici quelques sigles et abréviations à connaître et qui seront parfois utilisés par la suite dans ce livre :

- HTML : Hypertext Markup Language
- PHP : PHP, Hypertext PreProcessor
- SQL : Structured Query Language
- MySQL : Serveur de base de données et les outils pour y accéder
- LAMP : Linux - Apache - MySQL - PHP, le quatuor gagnant des serveurs web.

d. Pages dynamiques

Une page statique est une page écrite directement en HTML. Elle peut éventuellement incorporer du code Javascript lui donnant un semblant de 'dynamisme' mais uniquement du côté du navigateur et des données locales.

Pour des traitements plus lourds nécessitant l'accès à une base de données, un formatage de tableau en fonction de résultats, une recherche poussée, du graphisme, il faut passer par des pages dynamiques et par un langage qui sera exécuté du côté du serveur : ASP sur les serveurs Microsoft/IIS, Perl, PHP...

PHP est un langage Server Side ou côté serveur. Lors du chargement d'une page PHP, c'est le serveur qui va lire, interpréter et exécuter le code. Puis il renvoie le résultat au navigateur sous la forme de code HTML. Ainsi le navigateur et l'utilisateur ne voient jamais le véritable code PHP exécuté. De plus, le résultat étant une page web classique en HTML, pas besoin d'installer sur le client des composants spécifiques (java...). Il n'y a donc pas de notion de vitesse d'exécution du côté du client, mais celle du serveur est prépondérante.

e. Installer le nécessaire

PHP ne sert pas seulement à faire des pages dynamiques. C'est un langage interprété qui au même titre que Perl,

Python ou TCL est capable de lancer des scripts interactifs ou non. On peut même utiliser PHP pour créer des interfaces graphiques (extension GTK). Le minimum nécessaire et vital pour apprendre PHP est donc l'interpréteur PHP lui-même sur un environnement supporté (Unix, Windows, Mac...).

Si on parle de LAMP, on voit bien la référence. Un système Linux, un serveur Web Apache, un serveur de bases de données MySQL et un interpréteur PHP. La version 5 de PHP permet de fonctionner avec IIS et MySQL existe aussi pour Windows. On peut donc utiliser une architecture WIMP (Windows, IIS, MySQL, PHP) ou WAMP (Windows, Apache, MySQL, PHP).

Les distributions Linux disposent déjà du nécessaire pour l'architecture LAMP et le paramétrage est assez simple. De plus, le mode "user_mod" de Apache permet à chaque utilisateur de créer son site web dans un répertoire spécifique de son répertoire personnel. Vous vous reporterez à la documentation spécifique de votre installation pour savoir où placer vos scripts PHP afin de les tester. Sous Linux, c'est généralement dans **/var/www/htdocs**, ou **/srv/www/htdocs**, ou encore dans le répertoire **public_html** de votre répertoire personnel.

Sous Windows, c'est un peu plus complexe à mettre en œuvre manuellement. On emploie donc des distributions toutes faites, les meilleures se nomment EasyPHP et XAMPP. Elles proposent une interface graphique permettant de contrôler le démarrage des composants (PHP, Apache, etc.). On peut ainsi développer son site sous Windows et le porter tel quel sur un serveur Unix. Notez que XAMPP existe pour Windows, Linux, MacOS et Solaris.

Vous trouverez ces produits aux adresses suivantes :

- XAMPP : http://www.apachefriends.org/fr/xampp.html
- EasyPHP : http://www.easyphp.org/

Pour développer, il suffit d'un simple éditeur de texte mais il vaut mieux préférer un éditeur plus évolué supportant la coloration syntaxique et quelques fonctions évoluées. Sous Windows, les éditeurs notepad++ ou HAPEdit 3 sont parfaits pour développer en PHP. Sous Linux, kwrite, gedit, bluefish ou quanta sont parfaits.

Pour les tests, un simple navigateur web ou la ligne de commande suffisent.

f. Un premier programme PHP

Le premier programme PHP que vous allez taper, compiler et lancer est le fameux "Hello World" dont l'algorithme ne mérite évidemment pas d'être expliqué, car le programme se contente d'un simple affichage. Le code PHP résultant est le suivant.

```
<?php
echo "Hello world !" ;
?>
```

- Tapez ce programme et placez-le dans un fichier appelé **HelloWorld.php**.
- Ouvrez une fenêtre MS-DOS (cmd) sous Windows, ou une console (ou terminal) shell sous Unix/MacOS et tapez l'instruction suivante là où votre programme est enregistré. Le signe " > " indique l'invite de commande ou prompt de MS-DOS ou du shell, ne le tapez pas.

```
> php HelloWorld.php
Hello world!
```

Bravo, vous venez de faire fonctionner votre premier programme PHP.

La syntaxe du langage PHP de ce premier programme est très simple. Une page php porte l'extension **.php**. Une page PHP peut être entièrement programmée en PHP ou mélangée avec du code HTML. PHP est un langage "Embedded HTML", c'est-à-dire qu'il apparaît à n'importe quel endroit de la page HTML. Pour cela, on le place dans des balises particulières **<?php** et **?>**. On écrit donc une page HTML dans laquelle on intègre du code PHP. Le code PHP est encapsulé dans le HTML : c'est l'encapsulation.

```
<html>
  <head>
    <title>Titre</title>
  </head>
  <body>
    <?php
      echo "Hello World !";
    ?>
  </body>
</html>
```

Vous pourrez, selon votre configuration, accéder à la page Web via l'URL suivante :

```
http://localhost/HelloWorld.php
```

Le code HTML généré sera le suivant :

```
<html>
  <head>
    <title>Titre</title>
  </head>
  <body>
    Hello World !
  </body>
</html>
```

Pour la suite de cet ouvrage, quand ce sera possible, les exemples en PHP seront adaptés à une utilisation via un navigateur Web. Autrement, et ceci sera précisé, les programmes seront exécutés via la ligne de commande.

La variable

1. Principe

Vous savez grâce au chapitre précédent comment l'ordinateur se représente les chiffres et les nombres : sous forme de binaire. De même la mémoire de l'ordinateur, composée de cases, peut contenir des informations, notamment ces fameux nombres. En programmation, il faut quelque chose de simple, pratique et souple à manipuler pour représenter ces nombres.

Chaque case de la mémoire est numérotée. Si la mémoire fait 256 octets et que chaque case peut contenir un octet, alors il y a 256 cases numérotées de 0 à 255. On peut donc obtenir la valeur d'une case depuis son numéro, en disant que la case 74 contient la valeur 212. Là où cela se complique, c'est que la mémoire de vos ordinateurs atteint un nombre très important de cases. Avec 1 Go de mémoire, vous avez 1073741824 cases pouvant contenir chacune un octet. Comment voulez-vous vous souvenir de chaque numéro de case ? C'est bien entendu impossible.

Si par contre vous donnez un nom ou une étiquette à chaque valeur contenue dans la case, ou à une suite de valeurs de plusieurs cases, pour vous en rappeler plus facilement, cela devient bien plus évident. C'est ce qu'on appelle une variable. En informatique, une variable est l'association d'une étiquette à une valeur. Vous nommez la valeur. La variable représente la valeur et se substitue à elle. La variable est donc la valeur. Mais comme son nom l'indique, cette valeur peut changer dans le temps, soit que la variable ne représente plus la (ou les) même(s) case(s) mémoire, soit que la valeur de la case a changé.



Une variable est un nom ou étiquette donné à une valeur (nombre, texte, etc). Cette valeur peut varier au cours du temps : on affecte une nouvelle valeur au nom, d'où le nom de variable.

Quel nom donner à une valeur ? Le nom que vous voulez et qui, si possible est en rapport avec ce que représente la valeur. Ce peut être une lettre, une association de lettres et de chiffres, ou d'autres symboles. Le formalisme des noms des variables dépend du langage utilisé. Parfois, un caractère spécifique indique le type (que vous rencontrerez plus bas) de la variable : ce qu'elle peut contenir.

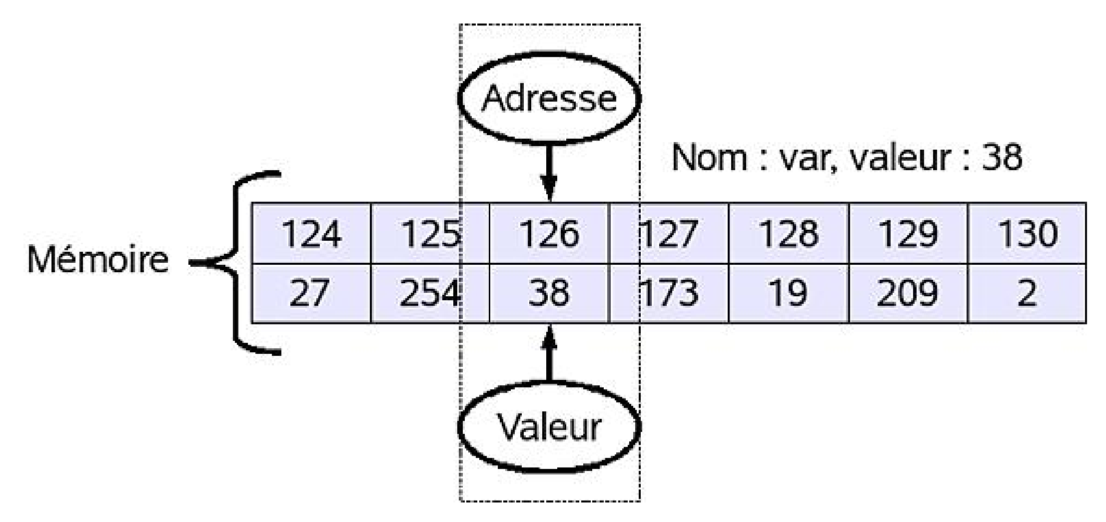
Certains langages acceptent des noms en lettres minuscules, majuscules, avec des chiffres dedans, des caractères spéciaux comme le souligné, etc. Quelques langages, dont Java, font la différence entre les minuscules et les majuscules. Si vous pouvez généralement utiliser un nom de grande taille, évitez pour des raisons purement pratiques les noms à rallonge. Voici quelques exemples de noms de variables :

- a
- var
- titre
- Total
- Somme_globale

Quand vous programmerez, vous veillerez à vous renseigner sur les conventions utilisées par le langage pour nommer vos variables : certains utilisent des syntaxes très particulières, d'autres sont beaucoup moins stricts.

La variable n'est qu'un outil pour les algorithmes et le programmeur, afin qu'il puisse se représenter "dans le réel" les données qu'il manipule. Si vous modifiez le nom "Somme_globale" par "Pomme_frite" partout dans l'algorithme ou le programme, la variable représentera toujours la même donnée, le programme fonctionnera à l'identique, mais ce sera plus difficile pour vous de faire le rapprochement. De même, la case mémoire ne porte pas réellement un nom ou étiquette. Chaque case est plutôt référencée par une adresse, elle-même exprimée par une valeur binaire. C'est le compilateur ou l'interpréteur qui fera la conversion des noms vers les adresses des cases mémoire à votre place.

Si vous souhaitez vous "amuser" à manipuler directement des adresses mémoire non pas par leur nom mais par leur adresse, quelques langages le permettent. Directement, vous pouvez apprendre l'assembleur (mais vous devrez être très courageux et patient), sinon des langages comme le C ou le C++ permettent la manipulation via des "pointeurs" : ce sont des étiquettes qui sont accolées à l'adresse de la case mémoire, contrairement aux noms des variables classiques qui sont associées à la valeur que la case contient. Comme en règle générale le nom d'une variable représente tant l'adresse que la valeur (la valeur a telle adresse mémoire), c'est source de beaucoup d'amusement (!) pour le programmeur...



La variable : une étiquette associée à une valeur en mémoire

2. Déclaration

Pour exister, une variable doit être déclarée, c'est-à-dire que vous devez indiquer au début de l'algorithme comment elle s'appelle et ce qu'elle doit contenir. En effet, pour que l'algorithme utilise votre variable, il doit déjà savoir qu'elle existe et ce qu'elle doit contenir. Il ne s'agit pas ici de définir la valeur de la variable, vous pourrez le faire dans la suite de l'algorithme en lui affectant une valeur. Il s'agit de donner son nom et de préciser le type de valeur qu'elle peut contenir. Les variables se déclarent au début de l'algorithme, avant le programme lui-même mais après le mot "**VAR**".

```
VAR
  Variable1 :type
  Variable2,variable3 :type
```

3. Les types

Une case mémoire contient généralement un octet, c'est-à-dire une valeur de 0 à 255. Mais une variable peut très bien contenir le nombre 214862, le réel 3,1415926, le texte "bonjour", etc. Donc une variable n'est pas uniquement définie par la valeur qu'elle contient, mais aussi par la place que cette valeur occupe et par la manière dont l'algorithme va la représenter et l'utiliser : nombre, texte, etc. C'est le **type** de la variable.

Que pouvez-vous mettre comme valeur dans une variable ? En principe, tout ce que vous voulez. Cependant, vous devez tout de même préciser quel type la valeur représente. Est-ce un nombre ?

Si oui, un entier (sans virgule) ou un réel (avec virgule) ? Est-ce du texte ? Est-ce un tableau ? Et ainsi de suite. Vous entendrez parfois parler du "codage" de la variable : selon le type et la taille de la valeur, celle-ci est encodée de manière différente dans la mémoire, utilisant plus de cases.

a. Les nombres

Placer des nombres dans la variable est le plus évident et souvent le plus courant. Une case mémoire peut contenir un octet, c'est-à-dire une valeur comprise entre 0 et 255 (2^8-1). Mais si elle doit contenir une valeur négative ? Alors sur les 8 bits, un sera réservé au signe et la case mémoire pourra contenir des valeurs de -127 à +128. On dit alors que la variable est "signée" : elle peut contenir des valeurs positives et des valeurs négatives. Seulement, 8 bits ne sont pas suffisants pour des grandes valeurs. C'est pourquoi les nombres peuvent être placés dans deux cases (16 bits), quatre cases (32 bits) ou plus.

Si l'ordinateur est bien plus à l'aise et bien plus rapide avec des nombres entiers, il sait aussi manipuler des nombres réels, bien que dans ce cas leur codage en binaire soit radicalement différent. Vous trouverez plus bas toutes les explications nécessaires pour comprendre comment l'ordinateur se représente exactement les nombres négatifs et réels : ce n'est pas si évident !

Au final, l'ordinateur est capable de gérer des nombres de longueur variable, signés ou non, entiers ou réels. Suivant le langage de programmation, les types portent des noms différents. Cependant le C et ses dérivés proposent les types suivants. Attention car Java fait la différence entre le type Byte de un octet et le type Char qui prend deux octets à cause du format de codage des caractères en Unicode.

Type numérique	Plage de valeurs possibles

Byte (char)	0 à 255
Entier simple signé (int)	-32 768 à 32 767
Entier simple non signé	0 à 65535
Entier long signé (long)	-2 147 483 648 à 2 147 483 647
Entier long non signé	0 à 4294967295
Réel simple précision (float)	Négatif : $-3,40x10^{38}$ à $-1,40x10^{45}$ Positif : $1,40x10^{-45}$ à $3,40x10^{38}$
Réel double précision (double)	Négatif : $-1,79x10^{308}$ à $-4,94x10^{-324}$ Positif : $4,94x10^{-324}$ à $1,79x10^{308}$

Sous PHP c'est encore plus simple. PHP est un langage non typé. Une variable peut contenir n'importe quelle valeur, de n'importe quel type, puis la même variable peut changer de contenu, d'un autre type. De plus, dans la plupart des cas la variable n'a pas besoin d'être déclarée. Elle se met à exister à sa première affectation, voire utilisation (si PHP est paramétré en ce sens). Dans ce dernier cas, l'accès à une variable qui n'a jamais eu de valeur retourne une valeur nulle, vide, fausse, selon le cas.

En PHP, il n'est pas tout à fait exact de dire que les variables ne sont pas typées. Elles le sont implicitement en fonction de leur contenu et de leur utilisation. Si vous affectez la valeur "32" à une variable, ce peut être un entier, mais aussi les deux caractères "32", selon le contexte.

Vous devez choisir quel type numérique utiliser selon vos besoins. La voie de la facilité consiste à prendre le type le plus élevé comme un entier long ou pire, un réel en double précision afin d'être tranquille. En informatique comme dans beaucoup de métiers, il faut choisir la formule la plus économique. On parle ici d'économie de moyens. Qu'un programme donne le résultat attendu n'est pas suffisant, il faut aussi qu'il le fasse vite, bien et en consommant le moins de ressources possibles. Le fait de disposer de plusieurs giga-octets n'est pas un critère suffisant pour programmer n'importe comment. Une liste de mille valeurs comprises entre 0 et 100 "coûtera" 1000 octets (soit pas loin de 1 ko) dans un type byte, mais 8000 octets (pas loin de 8 ko) soit 8 fois plus dans une type réel à double précision. Cela peut sembler faible, mais non seulement l'ordinateur doit manipuler 8 cases mémoire au lieu d'une, mais en plus il doit en permanence convertir une valeur qu'il pense être un nombre réel avec des chiffres après la virgule en entier, alors que c'est déjà le cas ! Quand vous verrez plus bas la formule mathématique nécessaire, vous vous rendrez compte du gâchis !

Utilisez les types qui vont bien avec les valeurs qui vont bien. En algorithmique, c'est bien plus simple. Rien ne vous empêche de préciser que vous manipulez des entiers ou des réels, mais vous pouvez aussi indiquer tout simplement que vous utilisez des variables contenant des valeurs numériques avec le pseudo-type "numérique". Vous devrez cependant être plus circonspect quand vous convertirez votre algorithme en vrai langage de programmation. D'une manière générale, deux types numériques sont utilisés en algorithmique :

- Les **entiers** : nombres sans virgule, négatifs ou positifs ;
- Les **réels** : nombres à virgule, positifs ou négatifs.

Les variables se déclarent toutes au début de l'algorithme. Si durant la rédaction de celui-ci vous remarquez que vous en avez besoin d'autres, vous les ajouterez au début.

```
VAR
  montant:réel
  somme,moyenne:réels
  qte :entier
```

Pour les variables contenant des nombres réels, ces derniers s'écrivent avec une vraie virgule comme en français "3,14" ou avec le point décimal, cependant dans les langages de programmation, ce sera bien souvent le point qui sera utilisé "3.14".

En PHP, comme les variables ne sont pas explicitement typées, il faut pouvoir déterminer autrement leur contenu. Dans l'exemple suivant, diverses valeurs sont affectées à des variables. PHP dispose de méthodes (des fonctions, expliquées dans le chapitre sur les sous-programmes) permettant de connaître le type actuel ou possible du contenu de la variable. Si le programme affiche 1, alors la variable est du type testé, sinon elle ne l'est pas. Les variables

commencent toujours par le caractère $ ce qui permet très facilement de les distinguer.

```
<html>
  <head><meta/>
    <title>Types</title>
  </head>
  <body>
  <?php
    $entier8bits=127;
    $entier16bits=32767;
    $entier32bits=2147483647;
    $entier64bits=9223372036854775807;
    $reel32bits=3.1415927;
    $reel64bits=3.14159265358979323846264338327950288419771;

    echo is_int($entier8bits)."<br />";
    echo is_int($entier16bits)."<br />";
    echo is_float($entier16bits)."<br />";
    echo is_long($entier32bits)."<br />";
    echo is_long($entier32bits)."<br />";
    echo is_float($reel32bits)."<br />";
    echo is_int($reel32bits)."<br />";
    echo is_double($reel64bits);
?>
    </body>
</html>
```

b. Autres types numériques

Il existe d'autres types numériques moins utilisés, tout au moins sur les ordinateurs personnels ou dans des langages de programmation classiques, mais bien plus sur des moyens ou gros systèmes ou dans des bases de données. Le système BCD " *binary coded decimal* " pour décimal codé en binaire est utilisé principalement en électronique car il est assez simple à mettre en œuvre. En BCD, chaque chiffre est codé sur 4 bits : $0000_{(2)}=0_{(10)}$, $0001_{(2)}=1_{(10)}$, $0010_{(2)}=2_{(10)}$, $0011_{(2)}=3_{(10)}$ et ainsi de suite jusqu'à $1001_{(2)}=9_{(10)}$. Comme un ordinateur ne manipule que des octets (composés de 8 bits), il existe deux méthodes pour coder des nombres en BCD :

- soit ne mettre qu'un chiffre par octet et compléter le reste que par des 1 ou des 0, "*EBCDIC*" ;
- soit mettre deux chiffres par octet et ajouter un signe à la fin, "*packed BCD*".

Par exemple le nombre 237 :

```
11110010 11110011 11110111 en EBCDIC
00100011 01111100 en Packed BCD (le 1100 final est le +, 1101 pour le -)
```

Vous rencontrerez peut-être un jour le type "monétaire" pour gérer les sommes de même nom et notamment les règles d'arrondi, mais aussi et surtout une grande quantité de types permettant de gérer les dates, couramment exprimés sous le nom "**date**". L'ordinateur de type PC stocke les dates de diverses manières, la plus commune étant sous la forme d'un **timestamp**, une valeur signifiant le temps écoulé depuis une date précise. Ce timestamp est souvent celui du système Unix, qui représente le nombre de secondes écoulées depuis le 1er janvier 1970 à minuit UTC, exactement. C'est donc un entier et le langage doit fournir des instructions pour convertir cette valeur en date réelle.

c. Les caractères

Si un ordinateur ne savait manipuler que les nombres, vous ne seriez pas en train de lire ce livre, écrit à l'aide d'un traitement de texte (OpenOffice.org), qui lui manipule toute sorte de caractères : chiffres, lettres, caractères spéciaux, etc. Une variable peut aussi contenir des caractères. Suivant les livres et sites Internet, vous trouverez les types "Alphanumérique", "Caractère", "Chaîne", "String". Ainsi une variable peut stocker votre nom, une ligne complète de texte, ou tout ce que vous voulez qui nécessite une représentation alphanumérique. On appelle d'ailleurs une suite de caractères alphanumérique une chaîne de caractères.

Si vous devez représenter un seul caractère, utilisez le type "**caractère**". Pour une chaîne, utilisez le type "**chaîne**".

```
VAR
      texte:chaîne
```

```
car:caractère
```

Quelle place occupe une chaîne de caractère ? En principe, un caractère occupe un octet. À chaque valeur comprise entre 0 et 255 est associé un caractère. C'est le principe de l'ASCII "*American Standard Code for Information Interchange*", norme de codage des caractères la plus connue et la plus utilisée. La table ASCII, inventée par les américains, contient à l'origine les 128 caractères utiles pour écrire en anglais. Par défaut elle ne contient pas les accents. Or la table ASCII peut contenir 256 caractères. Les 128 autres (huitième bit à un) contiennent des caractères semi-graphiques et des caractères spécifiques à certaines langues, typiquement le français, pour les caractères accentués. On parle alors de page de code ou de charset. Ces pages font l'objet d'une normalisation, par exemple la norme ISO 8859-15 qui est la page d'Europe de l'Ouest, avec le caractère de l'euro "€". Dans la plupart des langages ce codage sur un octet suffit et ainsi une chaîne de caractères de 50 caractères occupe 50 octets.

En pseudo-code algorithmique, les chaînes de caractères sont placées entre guillemets pour deux raisons :

1. Éviter une ambiguïté entre les nombres sous forme de chaîne de caractères et les nombres au format numérique. Certains langages ne font certes pas directement la différence (c'est selon le contexte d'utilisation) mais avec d'autres c'est catastrophique. La suite de caractères 1,2,3 représente-t-elle le nombre 123 et stockée ainsi en mémoire sous forme binaire dans un octet de la mémoire, ou la chaîne "123" stockée ainsi en mémoire sous forme de codes ASCII, soit un pour chaque caractère ? Les guillemets évitent les erreurs d'interprétation.

2. Ne pas confondre le nom de la variable avec son contenu, notamment lors d'une affectation. Ainsi, quand vous affecterez un valeur à une variable, si celle-ci est entre guillemets, vous lui affecterez une chaîne de caractères, si c'est un nombre, ce sera ce nombre (sachant que le nom d'une variable ne peut pas être constitué uniquement de chiffres) et enfin si ce n'est ni une chaîne ni un chiffre, c'est une variable. Dans ce cas, la première variable recevra comme valeur celle de la seconde qui lui est affectée. Ne pas respecter ce principe est une cause d'erreur grave et néanmoins commune.

En PHP, une variable n'étant pas typée, vous lui affectez une chaîne directement. Il est intéressant de savoir qu'il est possible de concaténer deux chaînes avec le (.). C'est un opérateur. Ils sont étudiés à la section Opérateurs et Calculs de ce chapitre.

L'exemple suivant contient deux affectations et affiche le résultat concaténé :

```
<html>
  <head><meta/>
     <title>Chaines de caractères</title>
  </head>
  <body>
  <?php

     $texte="Hello World !";
     $text2="Bonjour les amis";

     $texte=$texte.$text2;
     echo $texte;
?>
  </body>
</html>
```

d. Le type booléen

Pour déterminer si une affirmation est vraie ou fausse, l'ordinateur doit se baser sur le résultat de cette affirmation. En informatique, on emploie plus volontiers la notion d'expression et d'évaluation de cette expression. Une expression peut être décrite comme étant tout ce qui peut fournir une valeur que l'ordinateur peut déterminer, stocker, évaluer. Par exemple, l'affirmation "a>b" selon laquelle a est supérieur à b. Si a vaut 3 et b vaut 2, l'affirmation est vraie. Si maintenant a vaut 1 et b vaut 2, l'affirmation est fausse. Dans les deux cas "a>b" est une expression qui vaut soit vrai, soit faux. C'est le cas le plus simple, mais aussi le plus courant et le plus pratique comme vous le verrez lors des tests et des conditions.

Comment l'ordinateur détermine-t-il ce qui est vrai et faux ? C'est le rôle, dans l'architecture de Von Neumann, de l'UAL. Sous quelle forme l'ordinateur se représente-t-il ce qui est vrai ou faux ? Sous forme numérique, comme toujours. Tout ce qui retourne un résultat différent de zéro (0) est considéré comme étant vrai, donc si le résultat vaut zéro (0) alors il sera considéré comme faux. Avant de considérer cette définition comme exacte, renseignez-vous tout de même car certains langages (comme l'interpréteur de commande Unix) font l'inverse ! Cependant dans des langages comme Java ou le C, 1 est vrai, 0 est faux.

Pour représenter les valeurs vrai et faux, il suffit de deux chiffres, 0 et 1. Combien faut-il de place pour stocker ces deux valeurs ? Un seul bit ! En pratique, les langages gèrent les booléens de plusieurs manières. Certains créent des "champs" de bits dans un même octet, d'autres utilisent un octet complet, etc. Cependant, plusieurs langages

proposent le type booléen, très pratique.

Dans la pratique, ces langages proposent des constantes (des variables qui prennent une valeur une fois pour toutes) spéciales pour représenter les valeurs vrai et faux :

- **TRUE** pour vrai
- **FALSE** pour faux

Ces constantes peuvent même être utilisées directement pour évaluer une expression. Telle expression est-elle vraie, telle autre est-elle fausse ? Suivant le langage, il faudra faire attention si les constantes existent et sont en minuscules ou majuscules.

```
VAR

    Test:booléen
```

PHP dispose ne dispose évidemment pas de type spécial pour les booléens mais accepte cependant deux valeurs : **true** et **false**. À l'exécution, les valeurs associées (1 pour true, rien pour false) sont affichées.

```
<html>
  <head><meta/>
    <title>PHP et booleens</title>
  </head>
  <body>
  <?php
    $b1=true;
    $b2=false;
    echo "Vrai : $b1<br />";
    echo "Faux: $b2<br />";
?>
  </body>
</html>
```

4. Affectation

a. Affectation de valeurs

Dans le programme

Pour donner une valeur à une variable, il faut passer par un processus d'affectation à l'aide d'un opérateur. En pseudo-code, on utilise le symbole d'affectation " ← ". À gauche de ce symbole, vous placez le nom de la variable, à droite la valeur. Vous trouverez aussi dans certaines représentations algorithmiques le " **:=** " issu du Pascal. Les deux sont équivalents et utilisables (mais évitez de les mélanger, pour s'y retrouver).

Voici quelques exemples d'affectations en pseudo-code.

```
PROGRAMME AFFECTATION
VAR
  a:entier
  b,c:réels
  titre:chaîne
  vrai:booléen
DEBUT
  a←10
  b←3,1415927
  c←12345
  titre←"ma première affectation"
  vrai←TRUE
FIN
```

Vous avez évidemment rencontré dans les programmes PHP précédents comment affecter une valeur à une variable. C'est le signe "=" qui est utilisé. L'emploi du signe "=" en pseudo-code n'a pas la même signification, ce que vous verrez un peu plus loin dans les opérateurs de comparaison.

Attention cependant à ne pas vous tromper entre le type de la variable et la valeur que vous lui affectez. C'est une

cause d’erreur fréquente, tant en pseudo-code algorithmique que dans un vrai langage. Dans certains cas, cela pourra marcher (dans le sens où l’ordinateur ne retournera pas forcément une erreur) mais le résultat ne sera pas celui attendu. Considérez l’exemple, pas correct, suivant :

```
PROGRAMME AFFECT2
VAR
  a:entier
  b:réel
  c:chaîne
DEBUT
  b←3,1415927
  a←3,1415927
  c←12345
FIN
```

L’exécution de ce pseudo-code provoque quelques surprises et des erreurs. Tout d’abord b reçoit la valeur de PI et, comme b est déclaré en réel (pour l’exemple, le type classique Numérique n’aurait pas été pertinent), c’est correct. Puis a reçoit la même chose. Or a est un entier ! Suivant les langages de programmation, vous obtiendrez soit une erreur, soit cela fonctionnera, mais pas parfaitement. La variable a étant un entier, il se peut que a ne contienne que la valeur entière de b, soit 3. Notez que certains langages autorisent une conversion explicite d’un type vers un autre. On parle de **transtypage**. Quant à la variable c, elle doit contenir une chaîne de caractères, délimitée par des guillemets, absents ici, ce qui provoquerait probablement une erreur.

Le code PHP suivant fonctionne parfaitement : le transtypage est accepté. Il consiste à indiquer entre parenthèses, avant la valeur à affecter, le type final de celle-ci. La valeur sera convertie, si possible, dans ce type avant d’être affectée. Le transtypage vers le bas (d’un type de grande taille vers une taille réduite) risque de provoquer une perte de précision, voire même d’un grand nombre d’informations. Dans cet exemple, la variable f_a est explicitement convertie en entier. PHP ne va donc conserver que la partie entière de f_a et i_a contiendra 3.

```
<html>
  <head><meta/>
    <title>Egalité avec transtypage</title>
  </head>
  <body>
  <?php

    $f_a=(float)(3.1415927);
    $i_a=(int)3.1415927;

    echo "$f_a $i_a";
?>
  </body>
</html>
```

Ce qui retourne :

```
3.1415927 3
```

Afin de ne pas vous faire taper sur les doigts par votre professeur d’algorithmique, précisez le bon type dès le début et évitez les affectations douteuses.

Dans la déclaration

Vous avez le droit de donner une valeur initiale ou par défaut à une variable lors de sa déclaration. Dans ce cas, vous devez utiliser l’opérateur d’affectation lors de la déclaration.

```
PROGRAMME DECLARE2
VAR
  a←10:entier
  b←3.5,c←8.2347:réels
  titre←"Mon titre":chaîne
vrai←VRAI:booléen
DEBUT
  Afficher a
FIN
```

Avec une valeur par défaut, la variable dispose déjà d’un contenu dès son initialisation et peut être directement utilisée.

b. Affectation de variables

Le principe est exactement le même sauf que cette fois vous ne mettez pas de valeur à droite mais une autre variable, ce qui a pour effet d'affecter à la variable de gauche la valeur de la variable de droite.

```
PROGRAMME AFFECT3
VAR
  a,b:entiers
DEBUT
  a←10
  b←a
FIN
```

Là encore, vous prendrez bien soin de ne pas mélanger les torchons et les serviettes en n'affectant pas des variables de types incompatibles. L'exemple suivant est évidemment faux.

```
PROGRAMME AFFECT4
VAR
  a:entier
  b :réel
DEBUT
  b←3.1415927
  a←b
FIN
```

Là encore, n'oubliez pas une éventuelle conversion dans un vrai langage et de déclarer correctement vos variables dans le bon type. L'exemple PHP suivant ne devrait pas vous poser de problèmes de compréhension.

```
<html>
  <head><meta/>
    <title>Egalités variées</title>
  </head>
  <body>
  <?php

  $a=10;
  $r1=3.1415927;
  $txt1="Hello World";

  $b=$a;
  $r2=$r1;
  $c=(int)($r2);
  $txt2=$txt1;

  echo "$r2<br />";
  echo "$c<br />";
  echo "$txt2<br />";

?>
  </body>
</html>
```

5. Saisie et affichage

Pour simuler l'affichage d'un texte ou d'une valeur sur l'écran, il faut utiliser la pseudo-instruction "**Afficher**" qui prend à sa suite une chaîne de texte ou une variable. Si vous mélangez du texte et des variables, séparez ceux-ci par des virgules. À l'affichage, les virgules seront remplacées par des espaces.

```
PROGRAMME AFFICHE
VAR
  a:entier
  texte:chaîne
DEBUT
  a←10
  texte←"Hello World"
  Afficher a
  Afficher texte
```

```
  Afficher "Bonjour les amis"
FIN
```

Pour inviter un utilisateur à rentrer au clavier une valeur, utilisez le mot **Saisir**. L'algorithme attendra alors une entrée au clavier qui sera validée avec la touche d'entrée. La valeur que vous saisissez sera placée dans la variable indiquée à la suite de "**Saisir**".

```
PROGRAMME SAISIE
VAR
  reponse:chaîne
DEBUT
  Afficher "Quel est votre nom ?"
  Saisir reponse
  Afficher "Vous vous appelez",reponse
FIN
```

Si vous devez saisir plusieurs valeurs à placer chacune dans une variable, vous pouvez utiliser plusieurs "**Saisir**", mais plus simplement placez les diverses variables à la suite d'un unique Saisir, séparées par des virgules. L'utilisateur devra alors saisir plusieurs valeurs (selon le langage final : les unes à la suite des autres séparées par des espaces, ou en appuyant sur la touche [Entrée] après chaque saisie).

```
PROGRAMME SAISIE_MULTIPLE
VAR
  nom,prenom:chaînes
DEBUT
  Afficher "Quels sont vos noms et prénoms ?"
  Saisir nom,prenom
FIN
```

Vous avez déjà remarqué depuis le premier chapitre qu'en PHP les exemples utilisent "echo" pour afficher quelque chose dans la console MS-DOS, dans le shell Unix/MacOS ou à destination d'une page HTML pour un navigateur.

La saisie via la console, si le script PHP est lancé depuis un terminal, est très simple :

```
<?php

  echo "Entrez un texte : ";
  $saisie=fgets(STDIN);

  echo $saisie;

?>
```

Via le navigateur web, la saisie est totalement différente, puisqu'il faut passer par les formulaires. L'exemple suivant, dont les principes seront repris de nombreuses fois par la suite, partout où cela sera possible et cohérent, montre comment utiliser des formulaires en HTML et récupérer les valeurs saisies en PHP. C'est la base même de la puissance de ce langage. Les structures de contrôle (if) sont expliquées dans le prochain chapitre. Ce programme, exécuté une première fois, affiche le formulaire de saisie. La seconde fois, il affiche le résultat.

```
<html>
  <head><meta/>
    <title>Saisie en HTML</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['txt'])) {
  ?>
    <form method="GET">
      Votre saisie : <input type="text" size="15" name="txt" /><br />
      <input type="submit" name="OK" /><br />
    </form>
  <?php
  } else {
    $txt=$_GET['txt'];
    echo "Vous avez saisi $txt<br />\n";
  }

?>
  </body>
```

```
</html>
```

6. Les constantes

Vous pouvez décider de donner une valeur à une variable et que cette valeur ne doit pas changer : elle doit rester fixe dans le temps et inaltérable, pour toute la durée du programme. Sa valeur doit rester constante. D'où son nom.

Une constante est une valeur, représentée tout comme une variable par une valeur, qui ne peut pas être modifiée après son initialisation. Elle est immuable. Un exemple de constante pourrait être la valeur de PI.

Une constante se déclare généralement avant les variables sous le mot-clé **CONST**. Elle est aussi d'un type donné. Certains langages de programmation passent parfois outre du type de la constante.

```
PROGRAMME CONSTANTE
CONST
  PI←3.1415927:réel
VAR
  R←5:entier
  Aire:réel
DEBUT
  Aire←2*PI*R
  Afficher Aire
FIN
```

Une constante s'utilise exactement comme une variable, sauf qu'elle ne peut pas recevoir de valeur. En PHP, une constante est déclarée avec un sous-programme " define ". La première valeur est le nom de la constante, la seconde est la valeur associée. Notez qu'il n'y a pas de $ devant le nom d'une constante, ni à sa déclaration, ni lors de son utilisation.

```
<html>
  <head><meta/>
    <title>Cercle avec constante</title>
  </head>
  <body>
  <?php
    define("PI",3.1415927);
    $r=5.2;

    $surface=PI*$r*$r;
    $perimetre=2*PI*$r;

    echo "$surface $perimetre";
?>
  </body>
</html>
```

Opérateurs et calculs

1. Les affectations

Le symbole d’affectation "←" fait partie d’une grande famille, celle des opérateurs. Comme son nom l’indique, un opérateur est utilisé pour et dans des opérations. Le symbole "←" est un opérateur d’affectation. Il existe plusieurs opérateurs qui servent aux calculs, affectations, comparaisons, rotations (de bits), groupages, etc.

2. Les opérateurs arithmétiques

Pour que les algorithmes puissent effectuer des calculs, il faut pouvoir au moins faire des opérations simples. Vous utiliserez pour cela les symboles suivants :

- **+** : addition
- **-** : soustraction
- ***** ou **x** : multiplication (il est plus facile d’écrire un x pour fois qu’une étoile)
- **/** : division
- **%** ou **mod** : modulo
- **DIV** : la division entière



Rappel : un modulo est le reste d’une division entière. Par exemple 15/2 vaut 7, mais il reste 1. On dit que 15 modulo 2 vaut 1.

Ces opérateurs sont dits binaires car ils s’utilisent avec deux valeurs : une avant le symbole et une après. Les valeurs avant et après peuvent être des données (de même type que la variable qui reçoit les résultats) ou des variables. Voici un exemple d’opérations dans un simple algorithme qui calcule la surface et le périmètre d’un cercle.

```
PROGRAMME CERCLE
VAR
  r, PI, surface,perimetre:réels
DEBUT
  PI←3,1415927
  r←5,2
  surface←PI * r * r
  perimetre←2 * PI * r
  Afficher surface,perimetre
FIN
```

Ici il n’y a que des multiplications. Vous pouvez déjà remarquer que vous avez parfaitement le droit de chaîner vos calculs et de mélanger les données et les variables. Simulez les deux calculs :

```
surface←PI * r * r
surface←3,1415927 * 5,2 * 5,2
surface←84.948666608

perimetre←2 * PI * r
perimetre←2 * 3,1415927 * 5,2
perimetre←32.67256408
```

En PHP :

```
<html>
  <head><meta/>
```

```
    <title>Cercle</title>
  </head>
  <body>
  <?php
    $pi=3.1415927;
    $r=5.2;

    $surface=$pi*$r*$r;
    $perimetre=2*$pi*$r;

    echo "$surface $perimetre";
?>
  </body>
</html>
```

Il est possible de grouper les calculs avec des parenthèses "(...)". Celles-ci influent sur la priorité des calculs. Vous vous rendrez compte en effet que les opérateurs ont des degrés de priorité différents. Par exemple, une multiplication est "plus forte" qu'une addition.

Prenez l'exemple suivant :

```
PROGRAMME PRIORITE
VAR
  x,y,z,total:entiers
DEBUT
  x←3
  y←4
  z←5
  total←x + y * z
  Afficher total
FIN
```

Que vaudra total ? Si vous effectuez le calcul de gauche à droite, vous obtenez 3+4=7, 7*5=35. Si vous faites ceci avec votre calculatrice, vous n'obtiendrez pas ce résultat car la multiplication a un ordre de priorité plus élevé que l'addition. L'ordinateur va d'abord faire 4*5=20, puis 3+20=23. Le résultat est donc 23. Si vous souhaitez indiquer à l'algorithme et donc ensuite, dans un vrai langage, une modification des priorités, vous devez utiliser les parenthèses.

```
PROGRAMME PRIO2
VAR
  x,y,z,total:entier
DEBUT
  x←3
  y←4
  z←5
  total←(x + y) * z
  Afficher total
FIN
```

Cette fois vous obtenez le résultat (3+4)*5, ce qui vaut 35.

Voici l'équivalent en PHP qui regroupe les deux cas :

```
<html>
  <head><meta/>
    <title>Priorites</title>
  </head>
  <body>
  <?php
    $x=3;
    $y=4;
    $z=5;

    $total=$x+$y*$z;
    echo "$total<br />";

    $total=($x+$y)*$z;
    echo "$total<br />";
?>
```

```
  </body>
</html>
```

Voici un simple algorithme pour calculer les résultats d'une équation du second degré. Une équation du second degré est de la forme :

$ax^2+bx+c=0$

Pour résoudre une telle équation, il faut calculer un "discriminant" sous la forme :

$\Delta=b^2-4ac$

Suivant la valeur du discriminant, les résultats varient :

- si **Δ**>0, il y a deux solutions
- si **Δ**=0, il n'y a qu'une seule solution
- si **Δ**<0, l'équation n'a pas de solution.

Pour l'exemple, l'algorithme part du principe que l'équation est volontairement posée comme ayant deux solutions. Il sera complété dans le prochain chapitre consacré aux tests. Les résultats d'une équation du second degré sont appelés les racines. Pour les calculer, il faut utiliser les opérations suivantes :

$$x_1=\frac{-b+\sqrt{\Delta}}{2a} \quad \text{et} \quad x_2=\frac{-b-\sqrt{\Delta}}{2a}$$

Pour la racine carrée, vous utiliserez dans l'algorithme la syntaxe "racine(x)" où racine est une fonction mathématique qui calcule la racine carrée de x. C'est un peu l'équivalent des fonctions d'un tableur, vous verrez dans ce livre comment créer vos propres fonctions.

```
PROGRAMME EQUATION
VAR
  a,b,c,delta,x1,x2 :réels
DEBUT
  a←3
  b←6
  c←-10
  delta←( b * b ) - ( 4 * a * c )
  x1←( -b + racine(delta) ) / ( 2 * a )
  x2←( -b - racine(delta) ) / ( 2 * a )
  Afficher "les résultats sont :"
  Afficher "x1=",x1
  Afficher "x2=",x2
FIN
```

En PHP, notez l'utilisation de fonctions mathématiques intégrées comme sqrt (racine carrée) :

```
<html>
  <head><meta/>
    <title>Equation du second degré</title>
  </head>
  <body>
  <?php

    $a=3;
    $b=6;
    $c=-10;
    $delta=($b*$b)-(4*$a*$c);
    $x1=(-$b+sqrt($delta))/(2*$a);
    $x2=(-$b-sqrt($delta))/(2*$a);
    echo "les résultats sont :";
    echo "x1=$x1<br />";
    echo "x2=$x2<br />";
?>
  </body>
```

```
</html>
```

Quelques langages admettent des opérateurs arithmétiques unaires, c'est-à-dire qui ne prennent qu'une valeur :

- `++x` incrémente de 1 la variable x
- `x++` idem mais après l'utilisation en cours
- `--x` décrémente de 1 la variable x
- `x--` idem mais après l'utilisation en cours

PHP accepte ces opérateurs. Cette écriture peut surprendre. Voici un exemple :

```
PROGRAMME UNAIRE
VAR
  a:entier
DEBUT
  a←1
  Ecrire ++a
  Ecrire a++
  Ecrire a
FIN
```

- Le premier affichage indique 2, la variable a est incrémentée avant son utilisation.
- Le deuxième indique 2 aussi, la variable a est incrémentée après son utilisation.
- Le dernier indique 3.



Les opérateurs + et - peuvent aussi être utilisés comme opérateurs unaires : placés avant un scalaire ou une variable, le signe "-" donnera l'opposé de cette valeur (- -1 est égal à 1).

3. Les opérateurs booléens

Les opérateurs ne permettent pas que de faire des calculs. Dans une expression, un opérateur peut aussi effectuer des évaluations de booléens. Vous avez vu que pour l'ordinateur tout ce qui est vrai est différent de 0, ce qui est faux valant 0. Comment alors faire si deux expressions sont l'une vraie, l'autre fausse, pour connaître la valeur des deux conjuguées ? Il faut utiliser des opérateurs booléens pour indiquer ce qui doit être considéré comme vrai : l'un ou l'autre, les deux en même temps, etc.

- L'opérateur **ET** indique que les deux expressions situées avant et après doivent être toutes les deux vraies pour que l'ensemble le soit aussi.
- L'opérateur **OU** indique que seule l'une des deux expressions, que ce soit celle située avant ou celle située après, doit être vraie pour que l'expression complète soit vraie aussi.
- Le **NON** est la négation. Si l'expression était vraie elle devient fausse et vice versa.

Les opérateurs booléens sont régis par la logique booléenne, du nom de l'inventeur non pas de la logique elle-même, mais des travaux de George Boole qui au XIXème siècle a restructuré toute la logique en un système formel (que l'on peut interpréter avec des mots, des phrases compréhensibles). Prenez les deux expressions : "Il fait beau et le soleil brille". La première expression "il fait beau" est vraie s'il fait vraiment beau. La seconde expression "le soleil brille" est vraie si le soleil brille vraiment. Si les deux expressions sont vraies, alors l'expression globale est vraie. Par contre : "Il a neigé et il fait beau". S'il a neigé, cette première expression est vraie. Cependant s'il ne fait pas beau, la seconde expression est fausse. L'ensemble est donc faux (dans le cas contraire, chaussez vos skis).

Les trois opérateurs logiques ET, OU et NON peuvent être simplement compris à l'aide de petits tableaux appelés parfois "**tables de vérité**". Exp1 et Exp2 sont des expressions booléennes vraies ou fausses. Par exemple l'expression a=1 est vraie si a vaut vraiment 1.

ET :

Exp1 \ Exp2 (ET)	Vrai (1)	Faux (0)
Vrai (1)	Vrai (1)	Faux (0)
Faux (0)	Faux (0)	Faux (0)

Table de vérité ET

Ce tableau est à comprendre ainsi : si Exp1 est vraie et Exp2 est vraie, alors l'ensemble est vrai. On dit plus généralement que Exp1 ET Exp2 est vrai. Dans le cas du ET, l'ensemble est vrai uniquement si les deux expressions sont vraies : l'une ET l'autre. Dans les autres cas, comme au moins l'une des deux est fausse, alors le résultat total est faux.

OU :

Exp1 \ Exp2 (OU)	Vrai (1)	Faux (0)
Vrai (1)	Vrai (1)	Vrai (1)
Faux (0)	Vrai (1)	Faux (0)

Table de vérité OU

Dans le cas du OU, au moins l'une des deux expressions doit être vraie pour que l'ensemble soit vrai. Seule une seule assertion est donc fausse, dans le cas où les deux expressions sont fausses toutes les deux.

NON :

Exp1	NON Exp1
Vrai (1)	Faux (0)
Faux (0)	Vrai (1)

Table de vérité NON

Le NON est très facile à comprendre, puisque l'expression booléenne est inversée : ce qui était vrai devient faux et vice versa.

Dans quel cas les opérateurs booléens sont-ils utiles ? Dans les expressions utilisées dans des conditions (exécuter une action selon tel ou tel critère) quand ces conditions sont multiples. Par exemple, si vous voulez exécuter une action uniquement si deux variables a et b sont vraies (contiennent autre chose que 0).

```
PROGRAMME ET1
VAR
  a,b:entiers
  result:booléen
Début
  a←1
  b←2
  result←a ET b
  Afficher result
Fin
```

Notez que l'algorithme ci-dessus ne vérifie pas si a vaut 1 et b vaut 2. Il effectue l'opération logique "a ET b". Comme les variables a et b sont toutes les deux différentes de 0, elles sont considérées comme vraies. Donc la variable result contient "VRAI" et s'affichera ainsi si le langage le permet, ou sous forme de la valeur 1.

4. Les opérateurs de comparaison

L'algorithme ci-dessus ne vérifie pas les valeurs des variables. Il faut pour cela utiliser d'autres opérateurs. Pour évaluer une expression, il faut parfois avoir besoin de comparer des valeurs. Comment savoir si un utilisateur de votre logiciel a répondu oui ou non à votre question ? Comment savoir si le chiffre saisi dans le jeu du lancer de dé correspond au bon résultat ? Vous allez devoir utiliser les opérateurs de comparaison. Il y en a plusieurs. Ceux-ci sont des opérateurs binaires : ils prennent deux valeurs, une avant et une après. Ces valeurs peuvent être soit des scalaires (entiers, réels, chaînes de caractère - selon le langage utilisé) directement, soit leur représentation sous forme de variable. L'ordinateur évaluera le résultat de cette comparaison sous forme booléenne : le résultat sera vrai ou faux.

Les langages réagissent différemment selon les comparaisons et les types des données comparées. Là encore, prenez garde à ne pas mélanger les types. Aussi, si en pseudo-code algorithmique les chaînes de caractères peuvent être comparées avec tous les opérateurs, prenez garde à l'interprétation qui en est faite par les langages. En C notamment (et entre autres) il faut passer par des fonctions spécialisées.

a. L'égalité

L'opérateur d'égalité s'écrit avec le signe "=" et sert à vérifier si les deux valeurs à droite et à gauche sont identiques, c'est-à-dire qu'elles ont la même valeur. Dans cet exemple, l'expression a=b est vraie, mais a=c est fausse.

```
PROGRAMME EGALE
VAR
  a,b,c:entiers
DEBUT
            a←5
            b←5
            c←10
            Afficher a=b
            Afficher a=c
FIN
```

Il ne faut surtout pas confondre, tant en algorithmique que dans les langages de programmation, l'opérateur d'affectation "←" et l'opérateur d'égalité "=". En mathématique et en langage courant, a=b peut avoir deux significations : soit a reçoit la valeur de b, soit on cherche à vérifier si a et b sont égaux. Laquelle est la bonne ? Dans un langage comme le BASIC, l'interprétation du signe "=" dépend du contexte. Avec une variable avant et hors contexte de condition, c'est une affectation. Dans une expression conditionnelle, c'est une comparaison. Dur de s'y retrouver ! C'est pour cela que certains langages comme C, C++, Java ou encore PHP utilisent l'opérateur d'égalité "==", deux fois égal, pour ne pas le confondre avec l'opérateur d'affectation "=".

Dans ces langages :

```
a=b=c
```

est une expression valable qui signifie que a reçoit la valeur de b qui elle-même reçoit la valeur de c. La variable b est affectée en premier, puis a. Les trois variables disposent de la même valeur à la fin.

```
a=b==c
```

est aussi une expression valable. Le "==" est prioritaire sur le "=" et b==c est faux car 5 et 10 sont différents. Faux vaut 0. La variable a reçoit le résultat de l'expression b==c, donc a reçoit 0. L'expression totale est fausse, elle vaut zéro. Si vous aviez eu :

```
c=a==b
```

La valeur de a est égale à celle de b, donc l'expression "a==b" est vraie et vaut 1. Donc c vaut 1 et l'expression est vraie. Nuance...

b. La différence

L'opérateur de différence est décrit par les symboles "**≠**" ou "**!**=" (point d'exclamation et égal) qu'il faut comprendre comme la négation (voir opérateur booléen) de l'égalité. Vous trouverez parfois "<>", comme équivalent de inférieur ou supérieur : si c'est inférieur ou supérieur, alors ce n'est pas égal. Attention, une expression "a!=b" est vraie si la valeur de a est différente de b.

```
PROGRAMME DIFF
VAR
    a,b:entiers
DEBUT
    a←10
    b←20
    Afficher a!=b
    Afficher NON(a=b)
FIN
```

Les résultats de cet algorithme sont identiques. Les valeurs de a et de b sont différentes. Dans le premier cas, "a!=b" est vrai. Dans le second cas, "a=b" est faux, mais la négation de ce résultat est vraie.

c. Inférieur, supérieur

Quatre opérateurs permettent de comparer des valeurs inférieures et supérieures, avec ou sans égalité :

- **<** : inférieur
- **≤** ou **<**= : inférieur ou égal
- **>** : supérieur
- **≥** ou **>**= : supérieur ou égal

La compréhension de ces quatre opérateurs ne doit pas poser de problème. Le résultat est vrai si la valeur de gauche est inférieure, inférieure ou égale, supérieure, supérieure ou égale à la valeur de droite.

```
PROGRAMME INFSUP
VAR
  a,b,c:entier
DEBUT
            a←10
            b←10
            c←20
            Afficher a<c
            Afficher a<=b
            Afficher c>b
            Afficher c>=c

            Afficher NON(c<=a)
            Afficher c>a
FIN
```

Les quatre premières expressions décrivent parfaitement les résultats attendus : elles sont toutes vraies. Les deux dernières sont fausses et parfaitement équivalentes. Si la valeur de c n'est pas inférieure ou égale à a, elle lui est forcément supérieure. C'est encore ici une propriété de l'algèbre de Boole.

5. Le cas des chaînes de caractères

Vous pouvez, en pseudo-code algorithmique, utiliser les opérateurs de comparaison avec des chaînes de caractères. La comparaison s'effectue alors en fonction de l'ordre alphabétique des chaînes de caractères. Cet ordre est établi en fonction de la numérotation des caractères dans la table ASCII ou la page Unicode. Dans cet exemple, txt2 est supérieur à txt1 dans le sens où dans un tri alphabétique des deux, le b est situé après le a.

```
PROGRAMME TXT
VAR
  txt1,txt2:chaînes
DEBUT
  txt1←"a"
  txt2←"b"
  Ecrire txt2>txt1
FIN
```

Ces opérateurs appliqués à des chaînes de caractères sont un bon exemple de ce qui peut se passer si par hasard vous vous trompez dans les types et les affectations. Dans l'exemple suivant, les deux chaînes "1111" et "2" sont comparées, ainsi que les deux entiers de même valeur. Lesquelles sont les plus grandes ?

```
PROGRAMME TXTCOMP
VAR
  x,y :entiers
  txt1,txt2:chaînes
DEBUT
  x←1111
  y←2
  Afficher x>y

  txt1←"1111"
  txt2←"2"
  Afficher txt1>txt2
FIN
```

Dans le premier cas, l'expression est vraie. Dans le second, elle est fausse.

Il est possible d'additionner des chaînes de caractères. Le résultat est la concaténation des deux chaînes. Il n'est par contre pas possible d'utiliser les autres opérateurs arithmétiques. Vous pouvez utiliser l'opérateur "**&**" ou "+" pour concaténer, cependant le premier est préférable.

```
PROGRAMME CONCAT
VAR
  txt1,txt2:chaînes
DEBUT
  Afficher "Comment vous appelez-vous ?"
  Saisir txt1
  txt2←"Vous vous appelez "&txt1
  Afficher txt1
FIN
```

Pour aller plus loin

1. Les nombres négatifs

Un nombre signé par exemple sur 8 bits, contient un bit réservé pour le signe. C'est en tout cas ainsi qu'on vous le présente pour plus de compréhension. Généralement c'est le bit de poids fort, le plus à gauche, qui sert pour le signe : à 0 le nombre est positif, à 1 il est négatif. Par exemple $-9_{(10)}$ devrait être représenté par $10001001_{(2)}$. Cette représentation pratique pour le lecteur ne l'est cependant absolument pas pour l'ordinateur. Additionnez -9 et 30, vous obtenez 21.

En binaire, 30 équivaut à 00011110. Le binaire s'additionne comme le décimal : 1+1=10 donc retenue de 1 et ainsi de suite :

```
 00011110 (30)
+10001001 (-9)
=10100111 (-39)
```

Il y a un problème ! Vous devriez obtenir 21 soit 00010101 ! C'est qu'en réalité un nombre négatif n'est pas représenté comme ceci. L'ordinateur "ruse" avec les manipulations binaires. L'astuce consiste à prendre le complément à un de la valeur binaire en valeur absolue (-9 => 9) et de lui ajouter un (on obtient au final un complément à deux). Le complément à un consiste à remplacer tous les zéros (0) par des un (1) et tous les 1 par des 0.

```
 11111111 (complément à un)
 00001001 (9)
=11110110 (tout est inversé)
+00000001 (+1)
=11110111 (équivaut à -9 représentation machine)
```

 Si vous additionnez un nombre avec son complément à deux, vous obtenez 0 (plus une retenue).

Maintenant, additionnez ce résultat à 30 :

```
 11110111 (équivaut à -9 représentation machine)
+00011110 (30)
=00010101 (21 - plus une retenue)
```

C'est gagné ! En pratique, le microprocesseur n'effectue pas tous ces calculs de conversion car il sait représenter nativement en interne toutes ces valeurs, matériellement.

2. La représentation des nombres réels

S'il est simple de se représenter un nombre entier en binaire, cela semble plus complexe avec un nombre à virgule. En effet, le principe même du binaire veut que chaque valeur représente une puissance de 2 en fonction de sa position de 0 à n, donc une valeur entière. En plus, les nombres réels n'ont jamais la même taille : plus ou moins de chiffres avant la virgule, plus ou moins après. Il faut prendre le problème à l'envers : ne serait-ce pas plutôt la virgule qui se déplace ?

Ensuite, est-ce possible de représenter un nombre réel sous forme de résultat d'une manipulation de nombres entiers ?

Prenez un exemple simple : 1,2.

$1,2=12x0,1=12x10^{-1}=12E-1$

En véritable notation scientifique, on écrit 1,2E0 soit $1,2x10^{0}$.

Voilà qui est très intéressant. Les nombres 12, 10 et 1 pourraient parfaitement être codés directement en binaire. Certains ordinateurs spécialisés, dits calculateurs, fonctionnent de cette manière. Vérifiez avec une valeur plus importante : 182,1957 :

$182,195=182195x0,001=182195x10^{-3}=182195E-3$

En véritable notation scientifique, on écrit 1,82195E2 soit $1,82195 \times 10^2$.

C'est le principe de la notation scientifique qu'il va falloir retenir mais en binaire, pas en décimal. Le microprocesseur ne manipule pas de puissances de 10, mais de 2. Aussi il faut trouver, selon le même principe, un moyen de transformer tout ceci en binaire. Dans la partie avant la virgule, ce sont des puissances de 2 positives (2^0, 2^1, 2^2, etc). Après la virgule, il faut passer en puissances de 2 négatives (2^{-1}, 2^{-2}, 2^{-3}, etc). La première partie ne pose aucun problème.

$182_{(10)}=10110110_{(2)}$

La seconde partie est plus délicate. Elle se base sur les puissances négatives de 2. Pour rappel mathématique $2^{-1}=1/2^1$, $2^{-2}=1/2^2$, etc.

```
0,195x2=0,390 <1, on pose 0, 0,0...
0,390x2=0,780 <1, on pose 0, 0,00...
0,780x2=1,560 >1, on pose 1, 0,001...
0,560x2=1,120 >1, on pose 1, 0,0011...
0,120x2=0,240 <1, on pose 0, 0,00110...
0,240x2=0,480 <1, on pose 0, 0,001100...
0,480x2=0,960 <1, on pose 0, 0,0011000...
0,960x2=1,920 >1, on pose 1, 0,00110001...
0,920x2=1,840 >1, on pose 1, 0,001100011...
0,840x2=1,680 >1, on pose 1, 0,0011000111...
0,680x2=1,360 >1, on pose 1, 0,00110001111...
0,360x2=0,720 <1, on pose 0, 0,001100011110...
0,720x2=1,440 >1, on pose 1, 0,0011000111101...
0,440x2=0,880 <1, on pose 0, 0,00110001111010...
0,880x2=1,760 >1, on pose 1, 0,001100011110101...
0,760x2=1,520 >1, on pose 1, 0,0011000111101011...
```

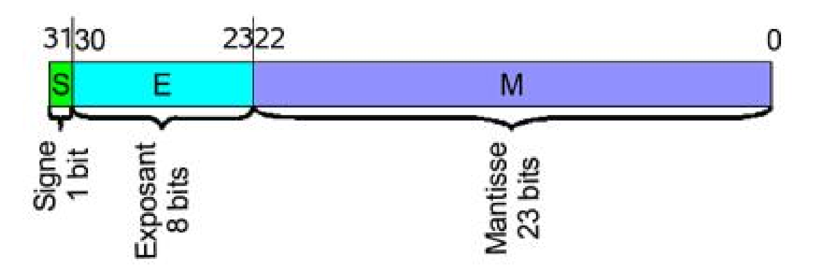
et ainsi de suite ! Ici vous obtenez une précision de 2^{-16}. Si d'ailleurs vous faites le total en décimal ($2^{-3}+2^{-4}+2^{-8}+2^{-9}+2^{-10}$...) vous obtenez un total de 0,1949920654296875. Vous voyez qu'on n'obtient pas la valeur exacte. Même avec plus de place et plus de calculs, le résultat approcherait 0,1949999999... sans jamais atteindre 0,195.

$0,195_{(10)}=0011000111101011_{(2)}$ arrondi à une précision de 2^{-16}.
$182,195_{(10)}=10110110,0011000111101011_{(2)}$, en arrondi.
$10110110,0011000111101011=1,01101100011000111101011 \times 2^7$

Enfin, puisque le 1 avant les virgules est implicite, on le supprime. On obtient ce qu'on appelle une mantisse.

Mantisse=01101100011000111101011

Plus on souhaite que la précision soit fine, plus on descend dans les puissances de 2. Vous voyez cependant qu'à un moment il faut s'arrêter et qu'il faut se contenter d'une précision de compromis. Ce système est aussi gourmand en place. Il existe une norme pour représenter les nombres réels en binaire, elle a été définie par l'IEEE, pour des précisions dites simples et doubles. En précision simple, le nombre réel est codé sur 32 bits. En précision double, il l'est sur 64 bits. Il existe aussi une précision sur 80 bits. Le principe est le même dans tous les cas. Il est basé sur la représentation du signe "S" du nombre, d'une mantisse "M" et d'un exposant "E".



Représentation binaire 32 bits d'un réel simple précision

Dans un nombre réel en simple précision sur 32 bits, un bit est réservé pour le signe, 8 pour l'exposant et 23 pour la

mantisse, dans cet ordre, le bit de poids fort étant le signe. L'exposant doit subir un décalage de 2^{n-1}-1, n étant le nombre de bits utilisés. Le décalage est donc de 127. Enfin, il ne faut pas conserver le 1 de la mantisse : il est implicite.

```
Signe S:0
Exposant E: 7+127=134(10), soit 10000111(2)
Mantisse: 182,195(10), sur 23 bits 01101100011000111101011(2)
```

Au final vous obtenez le nombre codé en 32 bits suivant :

S	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
0	1	0	0	0	0	1	1	1	0	1	1	0	1	1	0	0	0	1	1	0	0	0	1	1	1	1	0	1	0	1	1

Pour retrouver depuis ce nombre 32 bits la valeur réelle en décimal, il faut appliquer la formule suivante :

$$-1^S \times (1+\frac{M}{2^{23}}) \times 2^{E-127}$$

Formule de calcul d'un réel depuis sa représentation binaire

Reprenez les valeurs précédentes, converties en décimal pour plus de facilité :

- S=0
- E=137
- M=3551723

$$-1^0 \times (1+\frac{3551723}{2^{23}}) \times 2^{134-127}$$
$$\Leftrightarrow$$
$$(1+\frac{3551723}{8388608}) \times 2^7$$
$$\Leftrightarrow$$
$$182.1949920654296875$$

Le codage d'un réel sur 32 bits amène deux remarques :

- La taille de la mantisse limite la précision pour des nombres de grande valeur, car plus le nombre avant la virgule occupe de la place, plus la taille restante dans la mantisse est réduite pour les chiffres après la virgule. Cette remarque est aussi dans une moindre mesure valable pour un réel codé sur 64 bits.
- La précision peut se révéler insuffisante pour certaines applications. Si vous envoyez une fusée sur Jupiter en calculant la trajectoire en simple précision pour chaque élément de calcul intervenant, la fusée risque à cette échelle de ne pas arriver au bon endroit à cause des sommes d'imprécisions...

Le principe du réel en double précision est exactement le même, sauf que les tailles de l'exposant et de la mantisse sont agrandies. L'exposant fait 11 bits, la mantisse 52 bits. Si vous reprenez les calculs mais cette fois en double précision, vous arrivez à un total de 182,194999992847442626953125 avec un arrondi à 2^{24} !

Un exemple concret et simple des erreurs d'arrondi peut être tout à fait mis en évidence avec une simple calculatrice à très bas prix, ne serait-ce qu'en divisant 1 par 3. Obtenez-vous 1,333333 ou 1,333334 ? En réutilisant cette même valeur les calculs suivants sont souvent faussés.

Vu le nombre de manipulations pour arriver à gérer les nombres réels, leur manipulation par le microprocesseur est plus lente qu'avec des nombres entiers. Heureusement, à côté, puis dans les microprocesseurs sont apparus des composants supplémentaires appelés FPU, Flotting Point Unit, dont le but est de pouvoir manipuler directement les nombres et les fonctions mathématiques associées, accélérant fortement le traitement des données. Peut-être avez-vous entendu parler des processeurs Intel 386 qui équipaient les PC vers 1990. Un coprocesseur mathématique appelé 80387 pouvait souvent être ajouté pour accélérer les opérations. Même les ordinateurs personnels de type

Atari ST pouvaient se voir ajouter un coprocesseur de ce type (68881). Aujourd'hui ces coprocesseurs n'en sont plus : ils sont directement intégrés dans le microprocesseur. Le dernier microprocesseur de votre PC en contient.

3. Les dates

Les dates sont représentées de deux manières dans l'ordinateur. La première est le format BCD vu précédemment et ceci presque exclusivement dans le bios (setup) de votre ordinateur, pour des raisons historiques. Le second est le **timestamp** Unix, qui représente le nombre de secondes écoulées depuis le 1er janvier 1970 à minuit pile UTC. Cette date a été retenue car bien que l'idée d'Unix soit apparue en 1969, l'ère d'Unix (arrivée de la première version et expansion) débute dans les années 1970.

Le 11 avril 2007 à 21 heures, 24 minutes et 43 secondes, le timestamp Unix est de 1176319483 secondes.

Le timestamp est généralement codé sur un entier de 32 bits signé. S'il est négatif, il représente une date d'avant 1970, s'il est positif une date suivante. Il couvre une plage de 136 ans, du 13 décembre 1901 20 heures 45 minutes 52 secondes jusqu'au 19 janvier 2038, 3 heures 14 minutes 8 secondes. Passée cette date, que se passera-t-il ? Tous les systèmes d'exploitation ou les ordinateurs n'ayant pas prévu ce problème rencontreront un magistral bug digne de celui de l'an 2000, ou pire : le bug de l'an 2000 (qui n'a pas eu lieu, ou peu, malgré les prédictions catastrophiques) était essentiellement logiciel, le timestamp est défini au sein du système d'exploitation.

Alors que faire ? Déjà, il reste une trentaine d'années pour modifier la chose et c'est bien souvent déjà fait. Unix a été conçu il y a bientôt 40 ans et est toujours extrêmement utilisé. Il est fort probable que d'ici 30 ans, lui ou l'un de ses dérivés le soit encore. C'est un système d'exploitation qui a fait ses preuves. Or les technologies évoluent et les ordinateurs actuels manipulent sans difficulté des nombres de 64 bits. Un timestamp signé sur 64 bits permet de représenter des dates de l'époque où l'univers tel que nous le connaissons n'existait pas et, dans le futur, une période où notre Soleil n'existera plus depuis bien longtemps... Il suffit donc de passer le timestamp sur 64 bits et, sachant qu'il existe un type timestamp en programmation, de recompiler tous les programmes. Il n'y a donc aucun souci à se faire.

4. Les caractères

Les caractères sont habituellement codés sur 8 bits, soit un octet, en utilisant la table ASCII. Dans cette table, les 128 premières valeurs ne représentent pas que les caractères pour les anglais mais aussi divers codes non affichables utilisés pour contrôler l'affichage d'un terminal. Les caractères de 0 à 31 vont contenir des éléments comme le retour chariot, le passage à la ligne, la tabulation, etc. Le caractère 32 est l'espace, le 127 le caractère de suppression. Le "A" en majuscule démarre à 65, en minuscule à 97, les chiffres en partant de 0 à 48. Le reste est la ponctuation, divers signes comme les symboles monétaires, les opérateurs et comparateurs arithmétiques et ainsi de suite. Typiquement, la chaîne "Hello World!" est représentée ainsi.

Code ASCII	72	101	108	108	111	32	87	111	114	108	100	33
Caractère	H	e	l	l	o		W	o	r	l	d	!

Les caractères d'une même chaîne occupent tous des cases mémoires contiguës. L'ordinateur ou plutôt les langages utilisant les chaînes de caractères doivent savoir où celle-ci commence (c'est indiqué en interne par la variable elle-même) et où elle finit. C'est le caractère spécial de code 0, représentant un caractère vide, qui indique une fin de chaîne. Quand l'ordinateur doit récupérer une chaîne, il lit tous les caractères contigus jusqu'à ce qu'il trouve le caractère vide de code 0.

Les 128 caractères suivants représentent l'ASCII étendu, permettant de coder les caractères semi-graphiques qui étaient couramment utilisés sur les terminaux : traits horizontaux, verticaux, angles, etc, mais aussi les caractères propres à chaque pays. Ces 128 caractères sont placés au sein de pages de codes, dont la plupart sont normalisées. Quand on change de pays, on change de page de code. Ce système présente cependant deux inconvénients :

- Les fichiers textes et les noms de fichiers écrits dans d'autres langues que l'anglais ne seront pas interprétés correctement sur les systèmes n'utilisant pas la même page de code. Typiquement, si vous utilisez une page de code norvégienne sur des fichiers en français, vous obtiendrez des caractères surprenants.

- Les 128 octets ne suffisent pas toujours à coder tous les caractères d'une langue particulière, par exemple tous les idéogrammes chinois ou japonais. Dans ce cas, l'alphabet de ces langues est restreint, ou les gens natifs de ces pays (et d'autres) doivent utiliser plusieurs pages de code, ou passer par des logiciels et systèmes gérant spécifiquement leur langue.

Pour pallier ces inconvénients, il a fallu trouver un autre système de codage des caractères. L'arrivée des ordinateurs avec une grande capacité mémoire et une gestion avancée de l'affichage des caractères permet de disposer de polices de caractères (un fichier qui contient tous les dessins des caractères à destination de l'écran ou de l'imprimante) qui peuvent contenir les alphabets de la plupart des langues. Par exemple dans la police Arial, on

pourrait trouver les caractères européens, mais aussi hébreux, arabes, chinois, japonais et ainsi de suite. Mais comment représenter ces milliers de caractères différents en mémoire ?

Une norme appelée **Unicode**, reconnue par la plupart des systèmes d'exploitation et des logiciels notamment sous Unix et Windows, fait sauter cette limitation. Chaque caractère dispose d'un nom et d'un identifiant numérique, de manière unifiée et quelque soit le système cible. C'est-à-dire que tout produit sachant interpréter les caractères unicode affichera les chaînes de caractères écrites sous cette norme avec les bons caractères.

Un texte unicode en chinois s'affichera en chinois si votre traitement de texte gère l'unicode et dispose de la police de caractères unicode contenant les idéogrammes chinois. Notez que vous n'avez pas à vous soucier de la manière dont vous tapez le texte : le système d'exploitation et les logiciels le font à votre place : vous continuez à taper votre texte comme vous l'avez toujours fait.

Unicode est une norme de représentation interne des caractères et propose divers formats de stockage en mémoire. Actuellement, unicode représente plus de 245000 chiffres, lettres, symboles, ponctuation, syllabes, règles de représentation, etc. Il faut de la place pour représenter ceci. La méthode la plus courante, utilisée notamment par défaut sous Linux, est l'UTF-8, Universal Transformation Format. Son étude dépasse le cadre de ce livre et prendrait plusieurs pages. Cependant, sachez que le modèle Unicode est un modèle en couches.

La première couche est le jeu de caractères abstrait, en fait une liste des caractères et leur nom précis. Par exemple le "Ç" correspond à "Lettre majuscule latine c cédille".

La seconde est l'index numérique du caractère codé, appelé point de code, noté U+XXXX où XXXX est en hexadécimal. Sans rentrer dans les détails, le "Ç" est représenté par le point de codage U+00C7. Il existe plusieurs niveaux ensuite. Java utilise un codage des caractères sous forme Unicode sur 16 bits. Le premier octet en partant de la gauche représente le jeu de caractère, le second le numéro de caractère dans ce jeu. C'est pourquoi le type caractère en Java utilise deux octets, alors qu'il n'en utilise qu'un seul en C.

Types et langages

1. Langages typés ou non

Quelques langages sont très souples avec les variables. Vous pouvez tout d'abord y mettre des nombres, puis du texte, puis de nouveau des nombres. Ces langages sont dits "non typés". Certains poussent le raisonnement assez loin : une variable peut contenir le chiffre 3 et l'autre le texte "3 petits cochons", il pourra les additionner (ce qui devrait donner 6) ! C'est le cas du PHP par exemple : le type de la variable dépend alors de son contexte d'utilisation, le langage tentant de convertir son contenu quand c'est possible !

À l'inverse, d'autres langages sont de "typage fort" où toutes les variables doivent être déclarées de manière extrêmement précise : type, signe, longueur et les éventuelles conversions doivent être explicites.

En algorithmique, vous vous contenterez de donner le nom, le type et éventuellement la taille de la variable, qui gardera ses propriétés tout au long de l'algorithme, sa valeur pouvant bien entendu évoluer.

2. La gestion de la mémoire

La gestion de la mémoire est le calvaire des programmeurs en langages de bas niveau ou même de haut niveau quand ceux-ci laissent au programmeur la tâche de gérer la mémoire lui-même. C'est le cas de langages comme le C ou le C++. Imaginez une chaîne de caractères "Hello World!". Celle-ci est composée de 12 caractères en comptant la ponctuation et l'espace. Comme une chaîne se termine par un caractère nul, il faut, selon le principe qu'un caractère est codé en ASCII, 13 octets pour stocker cette chaîne en mémoire.

En algorithmique, vous n'avez pas à vous soucier de l'occupation mémoire de vos variables et chaînes. En Java, ou encore en PHP, non plus : ces langages disposent de mécanismes appelés "ramasse-miettes" qui le font pour vous. Mais en C par exemple, ce serait à vous de déclarer votre variable de manière à ce que son contenu puisse contenir jusqu'à 13 octets en déclarant, en fait, 13 cases mémoires d'un octet. Voici la méthode dite statique :

```
char texte[13] ;
```

ou encore la méthode dynamique :

```
char *texte=malloc(13*sizeof(char));
```

Ce n'est pas tout. En effet, avec cette dernière syntaxe le malheur veut que outre cette tâche complexe d'allocation mémoire, vous deviez libérer vous-même la mémoire, sinon votre programme continuera à consommer celle-ci jusqu'à la fin de son exécution. Mais ce système de gestion de la mémoire est-il vraiment un inconvénient ? Prenez en compte ces quelques allégations :

- La gestion de la mémoire de manière dynamique nécessite une connaissance avancée de la taille des variables utilisées, de la quantité de mémoire nécessaire et de la mémoire physique de la machine.

- L'accès à la mémoire passe par l'utilisation de variables particulières appelées pointeurs car elles ne représentent pas une valeur mais l'adresse d'une case. Une fois maîtrisées, celles-ci se révèlent être très puissantes et pratiques.

- L'allocation dynamique permet d'utiliser uniquement la quantité de mémoire nécessaire à un instant donné. C'est certes insignifiant pour quelques octets, mais s'il s'agit de manipuler de grosses images ou des films, cela compte.

- La mémoire inutilisée peut être libérée dès qu'elle n'est plus nécessaire. Avec les méthodes dites statiques, elle l'est uniquement à la fin du programme (ou d'un bloc d'instructions).

- L'allocation mémoire est la plus grande source d'erreurs dans un programme, pouvant occasionner du simple dysfonctionnement au plantage complet du programme, voire même de graves problèmes de sécurité (piratage) dans des applications critiques.

Partant du principe qu'un langage de haut niveau ne doit pas embêter le programmeur avec une quelconque gestion du matériel, PHP gère la mémoire à votre place et donc vous n'avez pas à vous soucier de libérer la mémoire de manière si complexe en apparence.

Les tests et conditions

1. Principe

Dans le précédent chapitre, vous avez pu vous familiariser avec les expressions mettant en place des opérateurs, qu'ils soient de calcul, de comparaison (l'égalité) ou booléens. Ces opérateurs et expressions trouvent tout leur sens une fois utilisés dans des conditions (qu'on appelle aussi des branchements conditionnels). Une expression évaluée est ou vraie (le résultat est différent de zéro) ou fausse. Suivant ce résultat, l'algorithme va effectuer une action, ou une autre. C'est le principe de la condition.

Grâce aux opérateurs booléens, l'expression peut être composée : plusieurs expressions sont liées entre elles à l'aide d'un opérateur booléen, éventuellement regroupées avec des parenthèses pour en modifier la priorité.

```
(a=1 OU (b*3=6)) ET c>10
```

est une expression tout à fait valable. Celle-ci sera vraie si chacun de ses composants respecte les conditions imposées. Cette expression est vraie si a vaut 1 et c est supérieur à 10 ou si b vaut 2 (2*3=6) et c est supérieur à 10.

Reprenez l'algorithme du précédent chapitre qui calcule les deux résultats possibles d'une équation du second degré. L'énoncé simplifié disait que pour des raisons pratiques seul le cas où l'équation a deux solutions fonctionne. Autrement dit, l'algorithme n'est pas faux dans ce cas de figure, mais il est incomplet. Il manque des conditions pour tester la valeur du déterminant : celui-ci est-il positif, négatif ou nul ? Et dans ces cas, que faire et comment le faire ?

Imaginez un second algorithme permettant de se rendre d'un point A à un point B. Vous n'allez pas le faire ici réellement, car c'est quelque chose de très complexe sur un réseau routier important. De nombreux sites Internet vous proposent d'établir un trajet avec des indications. C'est le résultat qui est intéressant. Les indications sont simples : allez tout droit, tournez à droite au prochain carrefour, faites trois kilomètres et au rond-point prenez la troisième sortie direction B. Dans la plupart des cas, si vous suivez ce trajet vous arrivez à bon port. Mais quid des impondérables ? Par où allez-vous passer si la route à droite au prochain carrefour est devenue un sens interdit (cela arrive parfois, y compris avec un GPS, prudence) ou que des travaux empêchent de prendre la troisième sortie du rond-point ?

Reprenez le trajet : allez tout droit. Si au prochain carrefour la route à droite est en sens interdit : continuez tout droit puis prenez à droite au carrefour suivant puis à gauche sur deux kilomètres jusqu'au rond-point. Sinon : tournez à droite et faites trois kilomètres jusqu'au rond-point. Au rond-point, si la sortie vers B est libre, prenez cette sortie. Sinon, prenez vers C puis trois cents mètres plus loin tournez à droite vers B.

Ce petit parcours ne met pas uniquement en lumière la complexité d'un trajet en cas de détour, mais aussi les nombreuses conditions qui permettent d'établir un trajet en cas de problème. Si vous en possédez, certains logiciels de navigation par GPS disposent de possibilités d'itinéraire Bis, de trajectoire d'évitement sur telle section, ou encore pour éviter les sections à péage. Pouvez-vous maintenant imaginer le nombre d'expressions à évaluer dans tous ces cas de figure, en plus de la vitesse autorisée sur chaque route pour optimiser l'heure d'arrivée ?

2. Que tester ?

Les opérateurs s'appliquent sur quasiment tous les types de données, y compris les chaînes de caractères, tout au moins en pseudo-code algorithmique. Vous pouvez donc quasiment tout tester. Par tester, comprenez ici évaluer une expression qui est une condition. Une condition est donc le fait d'effectuer des tests pour, en fonction du résultat de ceux-ci, effectuer certaines actions ou d'autres.

Une condition est donc une affirmation : l'algorithme et le programme ensuite détermineront si celle-ci est vraie, ou fausse.

Une condition retournant VRAI ou FAUX a comme résultat un **booléen**.

En règle générale, une condition est une comparaison même si en programmation une condition peut être décrite par une simple variable (ou même une affectation) par exemple. Pour rappel, une comparaison est une expression composée de trois éléments :

- une première valeur : variable ou scalaire
- un opérateur de comparaison
- une seconde valeur : variable ou scalaire.

Les opérateurs de comparaison sont :

- L’égalité : **=**
- La différence : **!=** ou **<>**
- Inférieur : **<**
- Inférieur ou égal : **<=**
- Supérieur : **>**
- Supérieur ou égal : **>=**

Le pseudo-code algorithmique n’interdit pas de comparer des chaînes de caractères. Évidemment, vous prendrez soin de ne pas mélanger les torchons et les serviettes, en ne comparant que les variables de types compatibles. Dans une condition une expression, quel que soit le résultat de celle-ci, sera toujours évaluée comme étant soit vraie, soit fausse.



L’opérateur d’affectation peut aussi être utilisé dans une condition. Dans ce cas, si vous affectez 0 à une variable, l’expression sera fausse et si vous affectez n’importe quelle autre valeur, elle sera vraie.

En langage courant, il vous arrive de dire "choisissez un nombre entre 1 et 10". En mathématique, vous écrivez cela comme ceci :

```
1 ≤ nombre ≤ 10
```

Si vous écrivez ceci dans votre algorithme, attendez-vous à des résultats surprenants le jour où vous allez le convertir en véritable programme. En effet les opérateurs de comparaison ont une priorité, ce que vous savez déjà, mais l’expression qu’ils composent est aussi souvent évaluée de gauche à droite. Si la variable nombre contient la valeur 15, voici ce qui ce passe :

- L’expression 1 ≤ 15 est évaluée : elle est vraie.
- Et ensuite ? Tout va dépendre du langage, l’expression suivante vrai ≤ 10 peut être vraie aussi.
- Le résultat est épouvantable : la condition est vérifiée et le code correspondant va être exécuté !

Vous devez donc proscrire cette forme d’expression. Voici celles qui conviennent dans ce cas :

```
nombre>=1 ET nombre<=10
```

Ou encore

```
1<=nombre ET nombre<=10
```

3. Tests SI

a. Forme simple

Il n’y a, en algorithmique, qu’une seule instruction de test, "**Si**", qui prend cependant deux formes : une simple et une complexe. Le test SI permet d’exécuter du code si la condition (la ou les expressions qui la composent) est vraie. La forme simple est la suivante :

```
Si booléen Alors
  Bloc d’instructions
FinSi
```

Notez ici que le booléen est la condition. Comme indiqué précédemment, la condition peut aussi être représentée par une seule variable. Si elle contient 0, elle représente le booléen FAUX, sinon le booléen VRAI.

Que se passe-t-il si la condition est vraie ? Le bloc d'instructions situé après le "**Alors**" est exécuté. Sa taille (le nombre d'instructions) n'a aucune importance : de une ligne à n lignes, sans limite. Dans le cas contraire, le programme continue à l'instruction suivant le "**FinSi**". L'exemple suivant montre comment obtenir la valeur absolue d'un nombre avec cette méthode.

```
PROGRAMME ABS
VAR
  Nombre :entier
DEBUT
  nombre←-15
  Si nombre<0 Alors
    nombre←-nombre
  FinSi
  Afficher nombre
FIN
```

En PHP, c'est le "if" qui doit être utilisé avec l'expression booléenne entre parenthèses. La syntaxe est celle-ci :

```
if(boolean) { /*code */ }
```

Si le code PHP ne tient que sur une ligne, les accolades peuvent être supprimées, comme dans l'exemple de la valeur absolue. Cet exemple montre une seconde possibilité par les mécanismes offerts par les bibliothèques de fonction de PHP.

```
<html>
  <head><meta/>
    <title>ABS</title>
  </head>
  <body>
  <?php
    $nombre=-15;
    if($nombre<0) $nombre=-$nombre;
    echo $nombre;

    echo "<br />";
  // seconde possibilitité
    $nombre2=-32;
    $nombre2=abs($nombre2);
    echo $nombre2;
  ?>
  </body>
</html>
```

b. Forme complexe

La forme complexe n'a de complexe que le nom. Il y a des cas où il faut exécuter quelques instructions si la condition est fausse sans vouloir passer tout de suite à l'instruction située après le FinSi. Dans ce cas, utilisez la forme suivante :

```
Si booléen Alors
  Bloc d'instructions
Sinon
  Bloc d'instructions
FinSi
```

Si la condition est vraie, le bloc d'instructions situé après le Alors est exécuté. Ceci ne diffère pas du tout de la première forme. Cependant, si la condition est fausse, cette fois c'est le bloc d'instructions situé après le Sinon qui est exécuté. Ensuite, le programme reprend le cours normal de son exécution après le FinSi.

Notez que vous auriez pu très bien faire un équivalent de la forme complexe en utilisant deux formes simples : la première avec la condition vraie, la seconde avec la négation de cette condition. Mais ce n'est pas très joli, même si c'est correct. Retenez que :

- Si dans une forme complexe, l'un des deux blocs d'instructions est vide, alors transformez-la en forme simple : modifiez la condition en conséquence.

- Laisser un bloc d'instructions vide dans une forme complexe n'est pas conseillé : c'est une grosse

maladresse de programmation qui peut être facilement évitée, c'est un programme sale. Cependant, certains langages l'autorisent, ce n'est pas une erreur ni même une faute lourde, mais ce n'est pas une raison…

L'algorithme suivant est une illustration de la forme complexe : elle vérifie si trois valeurs entrées au clavier sont triées par ordre croissant :

```
PROGRAMME TRI
VAR
  x,y,z:entiers
DEBUT
  Afficher "Entrez trois valeurs entières distinctes"
  Saisir x,y,z
  Si z>y ET y>x Alors
    Afficher " Triés en ordre croissant"
  Sinon
    Afficher "Ces nombres ne sont pas triés"
  FinSi
FIN
```

Le code suivant en PHP n'est pas bien difficile à comprendre.

```
<html>
  <head><meta/>
    <title>trie ?</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['x'])) {
  ?>
    <form method="GET">
      x : <input type="text" size="4" name="x" /><br />
      y : <input type="text" size="4" name="y" /><br />
      z : <input type="text" size="4" name="z" /><br />
      <input type="submit" name="OK" /><br />
    </form>
  <?php
  } else {
    $x=$_GET['x'];
    $y=$_GET['y'];
    $z=$_GET['z'];

    if($z>$y && $y>$x)
      echo "Tries en ordre croissant";
    else
      echo "Ces nombres ne sont pas tries";
  }
  ?>
  </body>
</html>
```

4. Tests imbriqués

Vous connaissez le dicton populaire "avec des si, on mettrait Paris en bouteille", qui répond généralement à une accumulation de conditions "s'il fait beau et qu'il fait chaud et que la voiture n'est pas en panne, alors nous irons à la mer, sinon si la voiture est en panne nous prendrons le train, sinon s'il y a grève alors nous ferons du stop, sinon si tout va mal alors nous resterons à la maison". C'est un peu lourd dit comme cela, mais qui n'a jamais échafaudé des plans douteux devant vérifier plusieurs conditions avec des scénarios de secours ?

Vous retrouverez parfois le même problème en programmation. Les deux formes de Si ci-dessus permettent de s'en sortir assurément, mais la syntaxe peut devenir très lourde. Vous pouvez en effet parfaitement imbriquer vos tests en plaçant des Si en cascade dans les blocs d'instructions situés après les Alors et les Sinon. Prenez l'exemple suivant : il faut deviner si un nombre saisi au clavier est proche ou non d'une valeur prédéfinie. Pour le savoir, le programme affiche froid, tiède, chaud ou bouillant suivant l'écart entre la valeur saisie et celle prédéfinie. Cet écart est calculé avec une simple soustraction, puis en déterminant sa valeur absolue.

```
PROGRAMME IMBRIQUE
```

```
VAR
  nombre,saisie,ecart :entiers
DEBUT
  Nombre←63
  Afficher "Saisissez une valeur entre 0 et 100 :"
  Saisir saisie
  ecart←nombre-saisie
  Si ecart < 0 Alors
    Ecart←-ecart
  FinSi
  Si ecart=0 Alors
    Afficher "Bravo !"
  Sinon
    Si ecart<5 Alors
      Afficher "Bouillant"
    Sinon
      Si ecart<10 Alors
        Afficher "Chaud"
      Sinon
        Si ecart<15 Alors
          Afficher "Tiède"
        Sinon
          Afficher "Froid"
        FinSi
      FinSi
    FinSi
  FinSi
FIN
```

Peut-être trouvez-vous cette syntaxe trop longue et surtout peu lisible. Une astuce consiste à tracer des traits verticaux allant des Si aux FinSi pour ne pas s'y perdre. C'est d'ailleurs recommandé par certains professeurs d'algorithmique, y compris pour les boucles (abordées dans un prochain chapitre). L'algorithme ci-dessus est parfaitement valable et d'une syntaxe correcte. Cependant, il est possible de faire plus concis avec la forme suivante.

```
Si booléen Alors
  Bloc d'instructions 1
SinonSi booléen Alors
  Bloc d'instructions 2
SinonSi booléen Alors
  Bloc d'instructions n
Sinon
  Bloc d'instruction final
FinSi
```

Cette forme est plus propre, plus lisible et évite de se mélanger les pinceaux. De multiples conditions sont testées. Si la première n'est pas vraie, on passe à la deuxième, puis à la troisième et ainsi de suite, jusqu'à ce qu'une condition soit vérifiée. Dans le cas contraire, c'est le bloc d'instructions final qui est exécuté. Les tests du précédent exemple doivent donc être réécrits comme ceci :

```
Si ecart=0 Alors
  Afficher "Bravo !"
SinonSi ecart<5 Alors
  Afficher "Bouillant"
SinonSi ecart<10 Alors
  Afficher "Chaud"
SinonSi ecart<15 Alors
  Afficher "Tiède"
Sinon
  Afficher "Froid"
FinSi
```

Voici le code correspondant en PHP :

```
<html>
  <head><meta/>
    <title>imbrique</title>
  </head>
  <body>
  <?php
```

```
  if(!isset($_GET['saisie'])) {
  ?>
    <form method="GET">
      Entrez un nombre entier : <input type="text"
size="4" name="saisie" /><br />
      <input type="submit" name="OK" />
    </form>
  <?php
  } else {
    $nombre=63;

    $saisie=$_GET['saisie'];

    $ecart=$nombre-$saisie;
    if($ecart<0) $ecart=-$ecart;

    if($ecart==0) echo "Bravo!";
    else if($ecart<5) echo "Bouillant";
    else if($ecart<10) echo "Chaud";
    else if($ecart<15) echo "Tiede";
    else echo "Froid";
  }
  ?>
  </body>
</html>
```

Quelle économie de place pour plus de clarté et de concision ! Maintenant vous disposez de tout le nécessaire pour résoudre une équation du second degré dans tous les cas de figure. Il manque deux cas :

- Si **Δ**=0, il n'y a qu'une seule solution qui vaut :

$$x_0 = \frac{-b}{2a}$$

- Si **Δ**<0, l'équation n'a pas de solution.

```
PROGRAMME EQUATION2
VAR
  a,b,c,delta,x1,x2:réels
DEBUT
  a←3
  b←6
  c←-10
  delta←( b * b ) - ( 4 * a * c )
  Si delta>0 Alors
    x1←( -b + racine(delta) ) / ( 2 * a )
    x2←( -b - racine(delta) ) / ( 2 * a )
    Afficher "Les deux solutions sont x1=",x1, "x2=",x2
  SinonSi delta=0 Alors
    x1← -b / (2 * a)
    Afficher "L'unique solution est :",x1
  Sinon
    Afficher "L'équation n'a pas de solution"
  FinSi
FIN
```

En PHP, le programme a été modifié légèrement pour autoriser la saisie de a, b et c :

```
<html>
  <head><meta/>
    <title>equation du second degre</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['a'])) {
```

```
  ?>
    y=ax*x+bc+c <br />
    <form method="GET">
      a : <input type="text" size="4" name="a" /><br />
      b : <input type="text" size="4" name="b" /><br />
      c : <input type="text" size="4" name="c" /><br />
      <input type="submit" name="OK" /><br />
    </form>
  <?php
  } else {
    $a=$_GET['a'];
    $b=$_GET['b'];
    $c=$_GET['c'];
    $delta=($b*$b)-(4*$a*$c);
    if($delta>0) {
      $x1=(-$b+sqrt($delta))/(2*$a);
      $x2=(-$b-sqrt($delta))/(2*$a);
      echo "Deux solutions :";
      echo "x1=$x1";
      echo "x2=$x2";
    }
    else if ($delta==0) {
      $x1=-$b/(2*$a);
      echo "Une seule solution x1=$x1";
    }
    else echo "Pas de solution";
  }
  ?>
  </body>
</html>
```

5. Choix multiples

Si les tests imbriqués facilitent parfois la vie, ils deviennent parfois trop lourds lorsque le nombre de tests devient trop important. Certains langages ont trouvé une intéressante parade à ce problème en proposant des structures de tests selon que telle expression est vraie, ou telle autre et ainsi de suite. Au lieu de faire des Si imbriqués ou des SinonSi, il suffit alors d'indiquer quoi faire quand telle ou telle valeur est rencontrée. En algorithmique, cela se traduit par la pseudo-instruction "**Selon que**".

```
Selon que :
  Condition 1 : bloc 1
  Condition 2 : bloc 2
  Condition n : bloc n
  Sinon : bloc final
Fin Selon
```

Une instruction "Selon que" peut être convertie facilement en Si imbriqués. Les conditions sont vérifiées les unes après les autres, dans l'ordre indiqué. Quand une condition est vraie, le bloc d'instructions associé est exécuté puis l'algorithme continue après le "Fin Selon". Si aucune condition n'est vérifiée, c'est le bloc final du "Sinon" qui est exécuté.

Voici une application simple qui permet de placer dans une variable le nom d'un mois en fonction de son numéro (entre 1 et 12) :

```
Variable mois en Numérique
Variable libelle_mois en Alphanumérique
Début
  mois←11
  Selon que :
    mois=1 : libelle_mois←"janvier"
    mois=2 : libelle_mois←"février"
    mois=3 : libelle_mois←"mars"
    mois=4 : libelle_mois←"avril"
    mois=5 : libelle_mois←"mai"
    mois=6 : libelle_mois←"juin"
    mois=7 : libelle_mois←"juillet"
    mois=8 : libelle_mois←"août"
    mois=9 : libelle_mois←"septembre"
```

```
    mois=10 : libelle_mois←"octobre"
    mois=11 : libelle_mois←"novembre"
    mois=12 : libelle_mois←"décembre"
  Fin Selon
Fin
```

En PHP la structure équivalente est le "**switch() ... case**".

```
switch(variable) {
  case valeur1 : <instructions> ; break ;
  case valeur2 : ... ; break ;
  ...
  default : <instructions> ;
}
```

Chaque valeur **case** correspond à une valeur possible de la variable du **switch**. Si plusieurs instructions sont présentes, il est préférable de les placer entre accolades. Le **break** est pour une fois utile et fortement conseillé. En effet si PHP tombe sur une correspondance, par exemple valeur1, et si aucun break n'est présent, alors il va exécuter toutes les instructions, jusqu'en bas, ou jusqu'à ce qu'il rencontre un break. En pratique, les instructions prévues pour valeur2, valeur3, etc, sont exécutées. Le **default** est l'action par défaut si aucune valeur **case** n'est vérifiée.

```
<html>
  <head><meta/>
    <title>bissextile</title>
  </head>
  <body>
  <?php
    $mois=5;

    switch($mois) {
      case 1: $libmois="Janvier"; break;
      case 2: $libmois="Février"; break;
      case 3: $libmois="Mars"; break;
      case 4: $libmois="Avril"; break;
      case 5: $libmois="Mai"; break;
      case 6: $libmois="Juin"; break;
      case 7: $libmois="Juillet"; break;
      case 8: $libmois="Août"; break;
      case 9: $libmois="Septembre"; break;
      case 10: $libmois="Octobre"; break;
      case 11: $libmois="Novembre"; break;
      case 12: $libmois="Décembre"; break;
      default: $libmois="???";
    }
    echo $libmois;
  ?>
  </body>
</html>
```

6. Des exemples complets

a. Le lendemain d'une date

Les structures déjà abordées permettent déjà d'effectuer pas mal de petites choses. Vous allez maintenant calculer le lendemain d'une date selon les critères suivants :

- La date est décomposée dans trois variables année, mois et jour.
- Il faut gérer :
  - les changements de mois
  - le nombre de jours dans le mois

- le changement d’année
- les années bissextiles pour le mois de février.

Pour information, une année est bissextile si elle vérifie intégralement deux règles :

- l’année est divisible par 4 et,
- l’année est divisible par 400 mais pas par 100.

L’algorithme pour indiquer si une année est bissextile ou non, est le suivant. Notez que dire qu’une année est divisible par n consiste à dire que le reste de la division par n est nul.

```
PROGRAMME BISSEXTILE
VAR
Annee :entier
DEBUT
  Afficher "Entrez l’année"
  Saisir annee
  Si (annee%4=0) ET ((annee%400=0) OU (annee%100>0)) Alors
    Afficher annee, " est bissextile."
  Sinon
    Afficher annee, " n’est pas bissextile."
  FinSi
FIN
```

En PHP :

```
<html>
  <head><meta/>
    <title>bissextile</title>
  </head>
  <body>
  <?php
  if(!isset($_GET[’annee’])) {
  ?>
    <form method="GET">
      Année : <input type="text" size="4" name="annee" />
      <input type="submit" name="OK" />
  </form>
  <?php
  } else {
    $annee=$_GET[’annee’];
    if(($annee%4==0) && (($annee%400==0) || ($annee%100>0)))
      echo "$annee est bissextile";
    else
      echo "$annee n’est pas bissextile";
  }
  ?>
  </body>
</html>
```

Ce test d’année bissextile n’intervient que lors des calculs sur le dernier jour de février, pour savoir si le lendemain du 28 est le 29 ou le 1er mars. De même, il faut gérer les cas où les mois ont 30 ou 31 jours, ainsi que le changement d’année lors du mois de décembre. L’algorithme utilise des structures "Selon Que" et " Si ".

```
PROGRAMME LENDEMAIN
VAR
  annee,mois,jour:entiers
DEBUT
  Afficher "Date initiale ?"
  Saisir jour, mois, annee
  Selon que:
    mois=1 OU mois=3 OU mois=5 OU mois=7 OU mois=8 OU mois=10:
      Si jour=31 Alors
        jour←1
```

```
        mois←mois+1
      Sinon
        Jour←jour+1
      FinSi
    mois=4 OU mois=6 OU mois=9 OU mois=11:
      Si jour=30 Alors
        jour←1
        mois←mois+1
      Sinon
        jour←jour+1
      FinSi
    Mois=2:
    Si (annee%4=0) ET ((annee%400=0) OU (annee%100>0)) Alors
      Si jour=29 Alors
        jour←1
        mois←mois+1
      Sinon
        jour←jour+1
      FinSi
    Sinon
      Si jour=28 Alors
        jour←1
        mois←mois+1
      Sinon
        jour←jour+1
      FinSi
    FinSi
    Mois=12:
      Si jour=31 Alors
        jour←1
        mois←1
        annee←annee+1
      Sinon
        jour←jour+1
      FinSi
  Fin Selon
  Afficher "Le lendemain est le ",jour, mois, annee
Fin
```

Le programme PHP associé reflète ce qui a déjà été expliqué ci-dessus : un **case** sans **break** continue l'exécution jusqu'au **break** suivant ou jusqu'à la fin. Aussi une suite de "case" sur la même ligne est possible : c'est comme si vous les mettiez les uns sous les autres. Pour le reste, le code est très proche de l'algorithme.

```
<html>
  <head><meta/>
    <title>Annees mois jours : lendemain</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['jour'])) {
  ?>
    <form method="GET">
      jour : <input type="text" size="4" name="jour" />
      <br />
      mois : <input type="text" size="4" name="mois" />
      <br />
      annee : <input type="text" size="4" name="annee" />
      <br />
      <input type="submit" name="OK" /><br />
    </form>
  <?php
  } else {
    $jour=$_GET['jour'];
    $mois=$_GET['mois'];
    $annee=$_GET['annee'];
    switch($mois) {
      case 1:case 3:case 5:case 7:case 8:case 10: {
        if($jour==31) {
          $jour=1;
```

```
          $mois++;
        } else $jour++;
          break;
      }
      case 4:case 6:case 9:case 11: {
        if($jour==30) {
          $jour=1;
          $mois++;
        } else $jour++;
        break;
      }
      case 2: {
        if( ($annee%4==0) && (($annee%400==0) || ($annee%100>0)) ) {
          if($jour==29) {
            $jour=1;
            $mois++;
          } else $jour++;
        } else {
          if($jour==28) {
            $jour=1;
            $mois++;
          } else $jour++;
        }
        break;
      }
      case 12: {
        if($jour==31) {
          $jour=1;
          $mois=1;
          $annee++;
        } else $jour++;
      }

    }
    echo "Demain est le $jour/$mois/$annee";
  }

  ?>
  </body>
</html>
```

b. La validité d'une date

Les calculs de dates sont une source inépuisable d'algorithmes. Sauriez-vous par exemple vous inspirer de l'algorithme ci-dessus pour tester la validité d'une date rentrée au clavier ? Il suffit pour chaque mois de vérifier si le jour entré est correct, le piège étant comme ci-dessus pour les années bissextiles et le mois de février. L'algorithme ci-dessous introduit une nouveauté : la présence d'un **drapeau** (qu'on appelle flag en anglais) ou indicateur. Le drapeau est une variable initialisée à une valeur prédéfinie au début du programme et dont la valeur est modifiée selon le résultat des tests. Après tous les tests, on vérifie la valeur du drapeau. A-t-elle changé ? Alors il y a eu un problème. Le drapeau est représenté par la variable "erreur". En fin de script, si elle contient autre chose que 0, alors la date est invalide. Sa valeur est modifiée à 1 lorsqu'un test n'est pas concluant.

```
Variables erreur, annee, mois, jour en Numérique
Début
  erreur←0
  Ecrire "Date initiale ?"
  Lire jour, mois, annee
  Si jour<0 OU mois<0 OU mois>12 Alors
    erreur←1
  Sinon Si Mois=1 OU mois=3 OU mois=5 OU mois=7 OU mois=8 OU mois=10
ou mois=12 Alors
    Si jour>31 Alors
      erreur←1
    FinSi
  Sinon Si mois=4 OU mois=6 OU mois=9 OU mois=11 Alors
    Si jour>30 Alors
      erreur←1
    FinSi
```

```
  Sinon SI mois=2 Alors
    Si (annee%4=0) ET ((annee%400=0) OU (annee%100>0)) Alors
      Si jour>29 Alors
        erreur←1
      FinSi
    Sinon
      Si jour>28 Alors
        erreur←1
      FinSi
    FinSi
  FinSi
  Si erreur=1 Alors
    Ecrire "Date incorrecte"
  Sinon
    Ecrire "Date correcte"
  FinSI
Fin
```

c. L'heure dans n secondes

Le simple est-il meilleur ?

Le but de l'algorithme est cette fois de déterminer l'heure qu'il sera dans n secondes. Pour cela, l'utilisateur doit saisir l'heure actuelle décomposée en heures (sur 24 heures), minutes et secondes. Bien entendu, il faut tenir compte du changement de jour, d'heure, de minute, sans oublier qu'il y a 60 minutes dans une heure et 60 secondes dans une minute. Par exemple, en additionnant 147 secondes à 23 heures, 58 minutes et 12 secondes, quelle heure sera-t-il ? Il sera minuit, 0 minute et 39 secondes le lendemain. Dans l'algorithme vous utiliserez des valeurs entières. Ainsi le résultat des divisions sera un entier et non un réel (par exemple 159/60=2,65 mais avec un entier vous récupérerez seulement 2).

- Dans un premier temps, additionnez les secondes : 147+12=159
- Ensuite, convertissez ces secondes en minutes. Pour cela, il suffit de diviser par 60 pour récupérer les minutes, puis de récupérer le reste de la division entière (le modulo) pour les secondes en trop (159/60 donne 2 minutes avec un reste de 39 secondes). Vous savez maintenant que la fin est de 39 secondes.
- Additionnez les minutes : 58+2=60. Si le chiffre est supérieur ou égal à 60, procédez comme pour les secondes. 60/60=1, soit une heure supplémentaire et un reste de 0 donc 0 minute. Vous savez maintenant que la valeur du milieu est 0 minute.
- Additionnez les heures : 23+1=24 soit minuit. Là encore, l'idéal est de compter les jours. Ainsi 24/24=1 (soit +1 jour), sans reste donc 0 heure : minuit.

Il sera donc minuit et 39 secondes.

```
PROGRAMME heure
VAR
  jours,heures,minutes,secondes,nbsec:entiers
DEBUT
  heures←17
  minutes←55
  secondes←48
  Afficher "Combien de secondes en plus ?"
  Saisir nbsec
  secondes←secondes+nbsec

  minutes←minutes+(secondes / 60)
  secondes←secondes % 60

  heures←heures+(minutes / 60)
  minutes←minutes % 60
  jours←heures / 24
  heures←heures % 24

  Afficher jours, heures, minutes, secondes
FIN
```

L'implémentation en PHP ne nécessite pas de commentaires particuliers. L'affichage final est juste un peu plus agréable.

```
<html>
  <head><meta/>
    <title>calcul d'heures</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['nbsec'])) {
  ?>
    <form method="GET">
      nb de secondes à ajouter : <input type="text"
size="4" name="nbsec" /><br />
    <input type="submit" name="OK" />
    </form>
  <?php
  } else {
    $heures=17;
    $minutes=55;
    $secondes=48;

    $nbsec=$_GET['nbsec'];

    $secondes+=$nbsec;
    $minutes+=(int)($secondes/60);
     $secondes=$secondes%60;

    $heures+=(int)($minutes/60);
    $minutes=$minutes%60;

    $jours+=(int)($heures/24);
    $heures=$heures%24;

    echo "$jours $heures $minutes $secondes";
  }
  ?>
  </body>
</html>
```

Les tests pour optimiser

Êtes-vous surpris par la forme que revêt cet algorithme ? Probablement car il n'y a aucun test d'effectué. La question se pose : sont-ils dans ce cas vraiment nécessaires ? La réponse ne coule pas de source. Rappelez-vous qu'il ne suffit pas qu'un algorithme fonctionne, mais qu'il fonctionne vite, bien et qu'il soit économe. Est-ce le cas ? Cet algorithme effectue neuf calculs : additions, divisions et modulos. Si on ajoute des tests, on ajoute des instructions et l'algorithme devient plus long. Or, vous aurez l'occasion de le voir dans les chapitres suivants, la complexité des algorithmes n'est pas liée à leur longueur. Certains sont très courts (comme celui-ci) et pourtant très gourmand en ressources. Inversement, d'autres sont longs et semblent compliqués pour un résultat de grande rapidité.

Un test bien posé peut éviter des calculs inutiles. Un calcul est gourmand en temps machine : le microprocesseur travaille plus longtemps en faisant des divisions et des modulos qu'en comparant deux valeurs : elles sont égales ou non et dans le cas de nombres, il suffit de faire un ET pour voir si cela retourne la même valeur, une opération élémentaire très rapide, bien plus que le moindre calcul. Autrement dit, vous avez tout intérêt à faire des tests quand ceux-ci peuvent permettre d'éviter des calculs lourds.

```
PROGRAMME HEURE2
VAR
  jours,heures,minutes,secondes,nbsec:entier
DEBUT
  heures←17
  minutes←55
  secondes←48
  Afficher "Combien de secondes en plus ?"
  Saisir nbsec
  secondes←secondes+nbsec
```

```
  Si secondes>59 Alors
    minutes←minutes+(secondes / 60)
    secondes←seconde % 60

    Si minutes>59 Alors
      heures←heures+(minutes / 60)
      minutes←minutes % 60

      Si heures>23 Alors
        jours←heures / 24
        heures←heures % 24
      FinSi
    FinSi
  FinSi
  Afficher jours, heures, minutes, secondes
FIN
```

Dans le meilleur des cas, un seul calcul sera effectué. Dans le pire, neuf. Entre les deux, toute une gamme. Si par exemple on additionne 60 secondes, alors on augmente forcément d'une minute et trois calculs supplémentaires sont effectués. Si on augmente d'une heure, alors trois autres nouveaux calculs ont lieu et d'une journée, deux derniers calculs. On retrouve bien neuf calculs, au pire. Mais la complexité moyenne est largement réduite par rapport au premier algorithme. Ce serait calculable via un intervalle aléatoire mais borné de n valeurs pertinentes.

Voici le programme modifié en PHP :

```
<html>
  <head><meta/>
    <title>Calcul d'heures 2</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['nbsec'])) {
  ?>
    <form method="GET">
      nb de secondes à ajouter : <input type="text"
size="4" name="nbsec" /><br />
      <input type="submit" name="OK" />
    </form>
  <?php
  } else {
    $heures=17;
    $minutes=55;
    $secondes=48;

    $secondes+=$nbsec;

    $nbsec=$_GET['nbsec'];

    if($secondes>59) {
      $minutes+=(int)($secondes/60);
      $secondes=$secondes%60;

      if($minutes>59) {
        $heures+=(int)($minutes/60);
        $minutes=$minutes%60;
        if($heures>23) {
          $jours+=(int)($heures/24);
          $heures=$heures%24;
        }
      }
    }

    echo "$jours $heures $minutes $secondes";
  }
  ?>
  </body>
</html>
```

Sachant que Unix compte le nombre de secondes écoulées depuis le 1er janvier 1970 à minuit pour calculer la date

actuelle, vous savez maintenant comment le système d'exploitation fait pour vous la fournir ! Et encore, il doit convertir les jours en années, en tenant compte des années bissextiles. Vous avez maintenant tout le nécessaire pour créer vous-même les morceaux d'algorithmes manquants. Vous pouvez même, si vous le souhaitez, créer un algorithme supplémentaire pour gérer par exemple les fuseaux horaires en fonction d'une heure centrale UTC. Il n'y a rien de difficile !

L'algèbre booléen

1. L'origine des tests

Les tests effectués tant en algorithmique qu'en programmation sont des tests logiques, ou plutôt faisant appel à la logique. Le chapitre Les variables et opérateurs a déjà brièvement abordé ce point lorsqu'il a été question des opérateurs booléens. Les opérateurs dits logiques ET, OU et NON sont des représentations de la logique. De quelle logique parle-t-on ?

Fondamentalement, la logique est la même pour tout le monde, bien que évidemment l'interprétation des résultats puisse varier d'un individu à l'autre (en statistique par exemple). La logique est universelle. Monsieur Spock, s'il avait existé n'aurait pas dit mieux. Mais jusqu'à peu (dans l'échelle de l'humanité) il n'y avait aucun moyen de se la représenter réellement, sous forme de symboles, d'assertions, etc. Il n'y avait aucune représentation formelle de la logique.

Or un ordinateur est logique (même si on peut lui demander des choses illogiques, c'est vous qui le programmez après tout). La logique est même la base de nombreuses applications mathématiques, électroniques, d'intelligence artificielle. En informatique, le matériel est électronique et dépend de la logique, les programmes dépendent tant de tests et de calculs faisant appel à la logique et devant fonctionner sur des circuits électroniques. Sans logique, pas d'électronique, ni d'ordinateurs, ni de programmes.

C'est ce qui fait que les opérateurs, conditions et tests ne doivent pas être posés n'importe comment. Il n'y a rien de plus logique qu'un ordinateur, mais aussi rien de plus stupide : il va bêtement (et encore la notion de bêtise lui est inconnue) exécuter exactement ce que vous lui demandez, même si le résultat entraîne une erreur ou est faux et cela du moment que les tests sont bien posés et qu'une réponse logique peut en être déduite. Ainsi :

```
PROGRAMME STUPIDE
VAR
  froid, nu, sortir en Booléen
DEBUT
  Froid←VRAI
  Nu←VRAI
  Si Froid=VRAI ET nu=VRAI Alors
    Sortir←VRAI
  FinSi
FIN
```

Cet algorithme peut être ainsi interprété : "S'il fait froid dehors et que je suis nu, alors je peux sortir". Cet algorithme est pour vous et moi, êtres humains, faux. Vous n'allez pas sortir nu s'il fait froid, vous auriez plutôt intérêt à mettre Sortir à FAUX, ou à inverser la condition froid ou nu. C'est évident ! Mais l'ordinateur s'en fiche : il n'a aucun moyen de savoir que vous avez fait une erreur dans vos tests. Le programme est mathématiquement logique : chaque test analysé est correct et donc les conditions sont remplies pour passer la variable Sortir à VRAI.

La suite aborde des points théoriques. Le but n'est pas de vous donner un cours magistral sur l'algèbre de Boole mais de vous fournir des bases de compréhension du fonctionnement de la logique vu du côté de l'informatique. Si vous voulez aller plus loin, il existe une littérature conséquente à ce sujet dans les bibliothèques et les librairies (par exemple, "Algèbre de Boole" chez Masson). S'il vous intéresse un jour d'aller largement au-delà et d'explorer les mécanismes de logique et de pensée humaine ou d'intelligence artificielle, il existe un gros ouvrage de référence, référence de nombreux scientifiques, informaticiens etc, qui s'appelle "Gödel, Escher, Bach : les brins d'une guirlande éternelle", par Douglas Hofstadter.

2. Petites erreurs, grosses conséquences

Comprenez-vous maintenant l'importance de la logique formelle et de bien écrire les tests et conditions dans un algorithme ?

Pour vous conforter un peu voici deux exemples de programmes mal écrits et de leurs désastreuses conséquences.

a. Ariane 5

Le 4 juin 1996 un bug d'un programme de la première fusée Ariane 5 a provoqué sa destruction après 40 secondes de vol. Le programme en question contrôlait les gyroscopes (qui indiquent l'orientation) de la fusée. Il venait d'Ariane 4 et n'avait pas été testé ni modifié pour Ariane 5. En gros, un nombre de 64 bits a été converti en 16 bits. Évidemment cela ne "rentre pas" et les valeurs retournées par ce programme sont devenus aberrantes. Or ce programme était critique et n'aurait jamais dû retourner de valeurs impossibles. Les données retournées n'étaient pas testées et vérifiées par le programme central de calcul de vol qui les prenaient pour argent comptant et les

interprétaient telles quelles. Sur un nombre signé, le dernier bit correspond au signe. Lorsque les 16 bits ont été remplis, le dernier bit est passé à un. Le programme a reçu une indication comme quoi la fusée avait changé de sens (pointait vers le bas) et a orienté les tuyères des réacteurs en les braquant à fond pour rectifier une situation totalement fausse. La fusée a donc subi à un moment donné de par la poussée et sa position, des forces aérodynamiques telles que sa destruction est devenue inévitable. Le comble ? Le programme des gyroscopes ne devait être utilisé que durant le compte à rebours et uniquement sur les modèles Ariane 3 ! Autrement dit, il n'aurait jamais dû être présent ni fonctionner en vol.

b. Mars Climate Orbiter

L'autre exemple touche encore au domaine spatial. Le 11 décembre 1998 la NASA lança une sonde en direction de Mars : Mars Climate Orbiter (MCO), dans le but d'étudier le climat martien. La sonde arriva en insertion orbitale (mise en orbite autour de Mars) le 23 septembre 1999. La sonde devait allumer son moteur principal un bon quart d'heure, passer derrière la planète (perte de contact avec la Terre) puis de nouveau devant (reprise de contact). Le contact n'a jamais repris. Pourquoi ? Parce que lors des calculs nécessaires à cette insertion sur orbite, MCO a fait appel à un programme issu d'un autre fournisseur (Lockheed Martin). Or MCO programmé par la NASA utilisait le système métrique (Newtons par seconde) et celui de Lockheed Martin le système de mesure impérial anglais (Livre par seconde). Personne n'a vérifié ni testé. Le résultat est que la sonde s'est approchée à très haute vitesse à 57 kms de la surface, au lieu de 150 kms et a brûlé dans l'atmosphère martienne. Bête erreur.

Imaginez maintenant les conséquences de ces deux très stupides erreurs de programmation (de simples tests de conversion), s'il y avait eu des astronautes dans ces missions… Heureusement, à votre modeste échelle, vous n'avez probablement pas la prétention d'envoyer des hommes sur Mars…

3. George Boole

Comme indiqué dans le chapitre Les variables et opérateurs, c'est l'anglais George Boole (1815-1684), logicien, mathématicien et philosophe qui le premier a posé les bases de la formalisation de la logique en proposant une analyse mathématique de celle-ci. En 1847 il publie un livre intitulé "*Mathematical Analysis of Logic*" (Analyse mathématique de la logique) puis en 1854 "*An Investigation Into the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities*" (une recherche sur les lois de la pensée, sur lesquelles sont fondées les théories mathématiques de logique et de probabilités). Dans ces ouvrages, George Boole développe une nouvelle forme de logique. Le formalisme rassemble une logique symbolique et une logique mathématique. Les idées doivent être traduites mathématiquement sous forme d'équations. Celles-ci peuvent ensuite être transformées via des lois et leurs résultats traduits en termes logiques.

Boole crée alors un algèbre basé sur deux valeurs numériques, le 0 et le 1, autrement dit le binaire, qui sera officialisé et mis en pratique dans le traitement de l'information le siècle suivant avec Claude Shannon.

Durant sa vie, les travaux de Boole restèrent au stade théorique. Cependant, son algèbre est à la base de nombreuses applications quotidiennes telles que l'électronique (les circuits logiques), l'informatique (processeurs, programmes), les probabilités, l'électricité (les relais), la téléphonie (les commutateurs), diverses recherches scientifiques, etc. Né très pauvre et autodidacte, George Boole finira membre de l'illustre Royal Society et effectuera de nombreux travaux sur les équations différentielles.

4. L'algèbre

a. Établir une communication

George Boole a développé l'algèbre qui porte son nom. Il est utilisé en mathématique, logique, électronique et informatique. Il permet d'effectuer des opérations sur les variables logiques. Comme son nom l'indique, il permet d'utiliser des techniques algébriques pour traiter ces variables logiques.

Prenez la phrase suivante absolument logique "Une proposition peut être vraie OU fausse, mais ne peut pas être vraie ET fausse". Autrement dit une variable logique ne dispose que d'un seul état à un moment donné : vrai ou faux. En prenant plusieurs propositions, on peut appliquer entre elles des formules algébriques.

La première utilisation de cet algèbre l'a été pour l'établissement de communications téléphoniques via la commutation téléphonique mise en place par Claude Shannon. Prenez cet exemple simple de mise en place d'une communication téléphonique entre deux interlocuteurs. Une communication nécessite à la fois un émetteur (qui émet l'appel) et un récepteur (celui qui décroche). On a donc :

```
Communication=Emetteur ET récepteur
```

La communication est établie si l'émetteur appelle et que le récepteur décroche. En d'autres termes, Communication est VRAIE si Emetteur est VRAI et récepteur est VRAI. Dans les autres cas, la communication ne s'établira pas et sera donc FAUX. Si vous appelez quelqu'un (VRAI) mais qu'il ne décroche pas (FAUX), ou encore que vous décrochez (VRAI) sans appel émis (FAUX), ou que personne n'appelle ni ne décroche (FAUX dans les deux cas), il n'y a aucune

communication d'établie. On peut en déduire la table suivante :

Emetteur	Récepteur	Communication
FAUX (n'appelle pas)	FAUX (ne décroche pas)	FAUX (pas de comm.)
FAUX (n'appelle pas)	VRAI (décroche)	FAUX (pas de comm.)
VRAI (appelle)	FAUX (ne décroche pas)	FAUX (pas de comm.)
VRAI (appelle)	VRAI (décroche)	VRAI (communication !)

Ne trouvez-vous pas que cette table ressemble beaucoup à l'opérateur logique ET ? Oui, c'est un exemple concret d'application de l'algèbre de Boole. Cette table où toutes les variables logiques sont posées avec leur résultat s'appelle une table de vérité.

Chaque case prend une valeur VRAI ou FAUX et la colonne finale le résultat attendu, lui-même VRAI ou FAUX. Remplacez VRAI et FAUX par les valeurs binaires respectives 1 et 0 :

Emetteur	Récepteur	Communication
0	0	0
0	1	0
1	0	0
1	1	1

Il y a des fois où il y a évidemment plus de deux variables logiques. Prenez l'exemple suivant. Quand décrochez-vous votre téléphone ? Quand il sonne ? Quand vous voulez appeler quelqu'un ? S'il sonne, voulez-vous vraiment répondre (si vous avez l'affichage du nom ou du numéro de l'appelant, vous voudrez peut-être filtrer vos appels) ?

Quels postulats posez-vous ? Vous décrochez si :

- Le téléphone sonne ET vous voulez répondre,
- Vous voulez appeler quelqu'un.

Autrement dit :

```
Décrocher=(Sonnerie ET volonté de répondre) OU envie d'appeler quelqu'un
```

Décrocher est VRAI si votre téléphone sonne (VRAI) ET que vous voulez répondre (VRAI) OU si vous voulez appeler quelqu'un (VRAI).

b. La vérité

Vous avez vu ci-dessus une table de vérité. Pour établir celle-ci, vous avez besoin de variables logiques qui ne prennent que deux valeurs : VRAI ou FAUX, qu'on appelle les **valeurs de vérité**. Ces valeurs forment un ensemble appelé B. VRAI et FAUX ne nécessitent que deux chiffres pour être représentés : `1` et `0`. L'ensemble B se note ainsi :

```
B={1,0}
```

En mathématiques, vous connaissez probablement d'autres ensembles, comme par exemple l'ensemble N des entiers naturels. Sur ces ensembles s'appliquent des lois, des théorèmes, des transformations. C'est pareil pour l'ensemble B, qui est régit par des lois et des transformations. De celles-ci peuvent être déduites une grande quantité de propriétés et de dérivations.

c. La loi ET

Vous la connaissez déjà depuis le chapitre Les variables et opérateurs et l'opérateur logique associé. La loi ET est aussi appelée la **conjonction**. Elle s'énonce ainsi :

a ET b est VRAI si et seulement si a est VRAI et b est VRAI

La loi ET utilise une notation particulière différente selon le champ d'application :

- "**.**" (le point) : a.b
- "∧" : a∧b
- "&" , "&&" ou "AND" en programmation, selon les langages.

La suite du chapitre utilisera la première notation avec le point. Cette loi est souvent associée à une vraie multiplication, car 0*n vaut toujours 0. Cependant attention, certaines propriétés ne s'appliquent pas du tout de la même manière !

La loi ET peut être décrite sous forme de table, à ne pas confondre avec une table de vérité. Elle ressemble plutôt à une table de multiplication...

Loi ET		
a\b	0	1
0	0	0
1	0	1

d. La loi OU

Vous connaissez aussi cette loi, rencontrée au chapitre Les variables et opérateurs avec l'opérateur logique associé OU. La loi OU est aussi appelée la **disjonction**. Vous trouvez parfois aussi "**disjonction inclusive**" afin de la différencier d'un autre opérateur. Elle s'énonce ainsi :

a OU b est VRAI si et seulement si a est VRAI ou b est VRAI

Notez que le OU étant inclusif (on y vient), si a et b sont VRAIS tous les deux, a OU b est VRAI aussi. Du moment qu'au moins l'un des deux est vrai, a OU b est VRAI.

Les notations suivantes sont utilisées :

- "+" (signe plus) : a+b
- "∨" : a∨b
- "|", "||", "OR" selon les langages de programmation.

La suite utilisera la première notation avec le signe " + ". La loi OU est souvent associée avec l'addition, ce qui se révèle totalement faux. En effet, 1+1 (addition binaire) vaut 0 avec une retenue de 1. Or, 1+1 est VRAI, donc 1… Attention au danger !

La loi OU peut être décrite sous forme de table.

Loi OU		
a\b	0	1
0	0	1
1	1	1

e. Le contraire

Le contraire, appelé aussi **négation** se définit ainsi :

Le contraire de a est VRAI seulement si a est FAUX.

Vous avez déjà rencontré le contraire avec l'opérateur NON. Le contraire est noté comme ceci :

- non-a
- ā
- ¬a
- "!", "NOT", "~" selon les langages de programmation.

Ainsi il suffit de retenir que ¬1=0 et que ¬0=1. Par la suite, ce sont les formes ā ou ¬ā qui seront utilisées, selon le cas (ex : ¬ā pour non non a).

f. Les propriétés

L'associativité

Elle est identique à l'algèbre classique. Dans certains cas, les parenthèses sont inutiles. Ainsi :

a+(b+c)=(a+b)+c=a+b+c

Et encore :

a.(b.c)=(a.b).c=a.b.c

La commutativité

Elle indique que l'ordre des variables logiques n'a aucune importance :

a+b=b+a

Et encore :

a.b=b.a

La distributivité

Attention, cette fois il y a une différence importante par rapport à l'algèbre classique et la distribution liée aux opérateurs + et *. C'est la raison pour laquelle il a été expliqué qu'il ne fallait pas les confondre avec les symboles logiques. En effet, si :

a.(b+c)=(a.b)+(a.c)

est identique,

a+(b.c)=(a+b).(a+c)

ne l'est pas du tout, mais est parfaitement correct en algèbre de Boole.

L'idempotence

L'idempotence signifie que si on applique la même opération une ou plusieurs fois alors on retrouve toujours le même résultat. Par exemple 3/1 vaut toujours 3, même si vous divisez n fois par 1. Cette propriété s'applique aux deux opérateurs. Ainsi :

a.a.a.a.a.a.a.a (etc)=a

et

a+a+a+a+a+a (etc)=a

La complémentarité

La négation de la négation d'une variable logique est égale à la variable logique. Ainsi la phrase " La vie est belle " équivaut à "La vie n'est pas non belle". En termes logiques a=non non a :

a=¬**ā**

De même :

a+¬a=1

En effet, l'expression "La vie est belle OU la vie n'est pas belle" équivaut à 0+1, ce qui est VRAI d'après la loi OU, l'une des variables au moins étant vraie. Enfin,

a.¬a=0

équivaut à dire que "La vie est belle ET la vie n'est pas belle", ce qui évidemment est impossible. D'après la loi ET, 1.0=0 donc FAUX.

La priorité

En calcul classique, si vous faites 1+2*3, vous obtenez 7 car la multiplication est prioritaire sur l'addition. En algèbre de Boole les priorités s'appliquent aussi. Le ET est prioritaire sur le OU. Vous pouvez cependant influer sur les priorités avec les parenthèses. Voici deux exemples. Dans les deux cas, a est FAUX (0), b est VRAI (1) et c est VRAI (1).

Exemple 1 :

a+b.c=?

- Le ET est prioritaire, on commence par b.c : 1.1=1
- Puis on passe au OU : 0+1=1
- On obtient donc a+b.c=VRAI

Exemple 2 :

a.b+c= ?

- Le ET est prioritaire, on commence par a.b : 0.1=0
- Puis on passe au OU : 0+1=1
- On obtient donc a.b+c=1, VRAI

Le théorème de De Morgan

Comme tout algèbre, celui de Boole dispose aussi de ses théorèmes. Le théorème de De Morgan établit deux vérités sympathiques qui sont souvent utiles pour réduire des calculs booléens et aussi dans l'établissement des tests et conditions. Ce sont en fait les propriétés associées au contraire (NON, négation). Prenez la table de vérité suivante :

a	**b**	**a+b**	**¬(a+b)**
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Prenez maintenant cette seconde table de vérité :

a	**b**	**¬a**	**¬b**	**¬a. ¬b**
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

Comparez les dernières colonnes et vous obtenez l'égalité suivante :

¬(a+b)=¬a.¬b

Dans les deux cas, le résultat sera VRAI uniquement si a **ET** b sont FAUX.

De la même manière, avec une table de vérité plus complète :

a	**b**	**a.b**	**¬(a.b)**	**¬a**	**¬b**	**¬a +¬b**
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	0	0	0	0	0

Comparez la colonne centrale avec la colonne finale et vous obtenez :

¬(a.b)=¬a+¬b

Dans les deux cas, le résultat sera VRAI seulement si a **OU** b sont FAUX.

g. Quelques fonctions logiques

Les fonctions logiques ne sont ni des lois ni des théorèmes : elles se déduisent de ces deux derniers sous forme de formules qui peuvent souvent être réduites, auxquelles on a donné un nom pour les raccourcir et pour plus de pratique. Elles sont souvent "câblées" en dur au sein des microprocesseurs et proposées par quelques langages de programmation.

Le OU exclusif XOR

Pas de jeu de mots ici, rien à voir avec la série japonaise. Dans le OU (appelé OU inclusif) le résultat est VRAI si a ou b ou les deux sont vrais. Dans le OU exclusif, le résultat est VRAI seulement si a ou b est vrai, mais pas les deux en même temps. Traduisez ceci en algèbre de Boole :

(a OU b) ET PAS (a ET b) soit (a+b).¬(a.b)

Si on développe on obtient :

- (a+b).(¬a+¬b) (le dernier terme provient du théorème de De Morgan)
- a.¬a+a.¬b+b.¬a+b.¬b
- a.¬a et b.¬b valent toujours 0 (FAUX), on les supprime, il reste
- a.¬b+¬a.b

Voici sa table de vérité :

a	**b**	**a⊕b**
0	0	0
0	1	1
1	0	1
1	1	0

Le OU exclusif est noté **XOR** (le X pour eXclusif). Vous le rencontrez sous ces notations :

- "≠" : différent de, en effet XOR est parfaitement équivalent

- "⊕" : un + entouré, a⊕b

Si le XOR n'est pas (ou peu) utilisé en algorithmique, beaucoup de langages de programmation le proposent, permettant de remplacer une condition longue par une condition plus courte. Les programmeurs n'ont souvent pas le réflexe de l'utiliser. La plupart des microprocesseurs intègrent directement une instruction de type XOR, accessible depuis le langage assembleur associé. Enfin, en électronique, les applications sous forme de porte logique sont nombreuses.

L'équivalence EQV

L'équivalence porte très bien son nom. Notée **EQV**, elle signifie que a EQV b est VRAI si et seulement si a et b ont la même valeur. En algèbre de Boole :

NON (a OU b) OU (a ET b) soit ¬(a+b)+(a.b)

Si on développe on obtient :

- (¬a+¬b)+(a.b) (le premier terme provient du théorème de De Morgan)
- (¬a+a).(¬a+b).(¬b.a).(¬b+b)
- (¬a+a) et (¬b+b) valent toujours 1 (VRAI), on les supprime, il reste
- **(¬a+b).(¬b+a)**

Voici la table de vérité de EQV :

a	**b**	**a⇔b**
0	0	1
0	1	0
1	0	0
1	1	1

Remarquez que l'équivalence est le contraire du XOR. Autrement dit :

a EQV b = ¬(a XOR b)

Si vous développez encore la négation de (¬a+b).(¬b+a) à l'aide des propriétés et du théorème de De Morgan, vous retrouvez la formule algébrique de XOR.

Le EQV est souvent représenté par le symbole "⇔" : a⇔b.

L'implication et l'inhibition

La plupart des langages ne proposent pas ces fonctions. L'implication indique que a est une condition suffisante pour b, tandis que b est une condition nécessaire pour a. Cela signifie que si b est vrai, ou bien que si a et b sont identiques, l'expression est toujours vraie :

L'implication se note a **IMP** b.

a IMP b = ¬a+b

a	**b**	**a⇒b**
0	0	1
0	1	1
1	0	0
1	1	1

L'inhibition est le contraire de l'implication, elle se note a **INH** b.

a INH b = a.¬b

a	**b**	**a⇍b**
0	0	0
0	1	0
1	0	1
1	1	0

h. Avec plus de deux variables

Rien n'empêche et, bien au contraire, d'utiliser plus de deux variables logiques dans des expressions de l'algèbre de Boole. Reprenez l'exemple de l'établissement de la communication téléphonique. Elle dispose de trois variables :

- sonner, qui sera appelée a,
- répondre, qui sera appelée b et
- appeler, qui sera appelée c.

Le résultat (décrocher) sera d, mais n'intervient pas dans les calculs. Quelle est la table correspondante ? Attention, il y a un piège. VRAI est représenté par 1, FAUX par 0.

a (sonner)	**b (répondre)**	**c (appeler)**	**d (décrocher)**
0	0	0	0
0	**0**	**1**	**1**
0	1	0	0
0	**1**	**1**	**1**
1	0	0	0
1	**0**	**1**	**1**
1	**1**	**0**	**1**
1	**1**	**1**	**1**

Avez-vous trouvé le piège ? Ça sonne, on n'a pas envie de répondre, mais on veut appeler (pour appeler). Dans ce cas, vous n'allez pas décrocher : vous attendrez que le téléphone arrête de sonner, or ci-dessus vous le faites quand même. Donc la table ci-dessus n'est pas correcte. Une ligne est fausse. Voici la bonne table :

a (sonner)	**b (répondre)**	**c (appeler)**	**d (décrocher)**
0	0	0	0
0	**0**	**1**	**1**
0	1	0	0
0	**1**	**1**	**1**

1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	**1**	**1**	**1**

Trouver l'expression minimale

La difficulté consiste maintenant à trouver comment, depuis cette table de vérité, définir une expression booléenne qui retourne VRAI. Vous constatez que seules quatre lignes de la table de vérité sont vraies, c'est-à-dire que la personne va vraiment décrocher.

Le résultat est à 1 quand a, b et c valent :

- 0, 0, 1
- 0, 1, 1
- 1, 1, 0
- 1, 1, 1

Vous pouvez aussi écrire que d est vrai quand (a,b,c)=(0,0,1) ou (0,1,1) ou (1,1,0) ou (1,1,1). Convertissez cet énoncé en expression booléenne :

```
d=¬a.¬b.c+¬a.b.c+a.b¬c+a.b.c
```

Il est intéressant de remarquer que les deux premiers termes de l'expression peuvent être factorisés par ¬a.c et les deux derniers par a.b selon la propriété de distributivité. Le résultat devient donc :

d=(¬a.c).(b+¬b)+(a.b).(c+¬c)

De même comme la propriété de complémentarité indique que a+¬a=1, les expressions b+¬b et c+¬c sont toujours vraies. Elles peuvent être supprimées. Le résultat final devient :

d=¬a.c+a.b

Ce n'est malheureusement pas toujours aussi évident. Avec quatre, cinq, six termes, les expressions deviennent beaucoup plus longues et complexes. Il faut toujours chercher à faire au plus simple, avec le moins de termes et de variables possibles, quitte à éclater les expressions booléennes.

Application dans l'algorithme

En algorithmique, l'expression booléenne précédente serait traduite ainsi dans un test :

```
PROGRAMME TELEPHONE
VAR
  a,b,c,d:booléens
DEBUT
  a←VRAI
  b←VRAI
  c←FAUX
  Si ((NON a) ET b) OU (a ET c) Alors
    d←VRAI
  Sinon
    d←FAUX
  FinSi
  Si d=VRAI Alors
    Afficher "Je décroche"
  Sinon
    Afficher "Je ne décroche pas"
  FinSi
FIN
```

De manière plus simple, rien n'empêche de faire ceci :

```
PROGRAMME TEL2
VAR
  a,b,c,d :booléens
DEBUT
  a←VRAI
  b←VRAI
  c←FAUX
  d←((NON a) ET b) OU (a ET c)
  Si d Alors
    Afficher "Je décroche"
  Sinon
    Afficher "Je ne décroche pas"
  FinSi
FIN
```

Le "Si d" équivaut à "Si d=VRAI".

5. Une dernière précision

Avec les conditions suivantes : "S'il fait chaud et qu'il ne pleut pas, alors je vais me promener", vous aurez parfois la tentation de faire ceci :

```
PROGRAMME CHAUD
VAR
  chaud,pleuvoir:booléens
DEBUT
  chaud←VRAI
  pleuvoir←FAUX
  Si chaud=VRAI Alors
    Si pleuvoir=FAUX Alors
      Afficher "Je me promène "
    Sinon
      Afficher "Je rentre"
    FinSI
  Sinon
    Afficher "Je rentre "
  FinSI
Fin
```

C'est possible, cela marche, mais ce n'est pas très beau. Deux tests, deux conditions et une répétition inutile. Pourquoi ne pas faire ceci ?

```
PROGRAMME CHAUDMIEUX
VAR
  chaud,pleuvoir :booléens
FIN
  chaud←VRAI
  pleuvoir←FAUX
  Si chaud=VRAI ET pleuvoir=FAUX Alors
    Afficher "Je me promène "
  Sinon
    Afficher "Je rentre "
  FinSI
FIN
```

Cela marche exactement de la même manière, mais il n'y a plus qu'un test et pas de répétition. C'est plus court. Tout de même, cet algorithme a plus de classe que le précédent ! Et pour épater un peu plus vos éventuels professeurs, pourquoi ne pas remplacer le test par :

```
...
Si chaud ET NON pleuvoir Alors

...
```

Les structures itératives

1. Définition

Comme indiqué dans le premier chapitre, la boucle est la quatrième grande structure de base de l'algorithmique et donc de la programmation. Passé ce chapitre, tout le reste est une application ou une dérivation de ces quatre structures de base. Les boucles sont des structures itératives. Une **itération** ou **structure itérative** est une séquence d'instructions destinée à être exécutée plusieurs fois. C'est aussi l'action d'exécuter cette instruction. Vous entendrez parler parfois de **structures répétitives**, c'est la même chose dite autrement. Le but d'une boucle est de répéter un bloc d'instructions plusieurs fois. Selon le type de boucle, ce bloc va être répété un nombre fixe de fois (n fois) ou selon un certain nombre de critères (un test de une ou plusieurs conditions) que vous connaissez très bien maintenant.

La boucle est un élément très simple au premier abord. Les premiers exemples que vous rencontrerez seront bien souvent évidents. Pourtant elle devient rapidement l'une des bêtes noires du programmeur en herbe à cause justement des fameux critères de sortie. Si les tests exécutent une action donnée (structure SI) en cas de réussite ou non, une erreur dans une condition de sortie peut amener au mauvais nombre de boucles, à ne jamais y rentrer ou même pire, à ne jamais en sortir.

La boucle est d'autant moins simple à assimiler qu'il est probable que vous n'ayez jamais rencontré une structure de ce genre hors de l'algorithmique et des langages de programmation. Dans le langage courant, on ne parle pas de boucle quand il s'agit de réciter une table de multiplication. En algorithmique vous devrez pourtant en utiliser une pour calculer cette table. De même dans l'utilisation quotidienne de l'ordinateur, vous n'utilisez pas cette structure pourtant, tous les programmes le font. Comment lire l'intégralité d'un fichier de traitement de texte ? Comment jouer un mp3 ou une vidéo ? À l'aide des boucles bien entendu !

2. Quelques usages simples

Un exemple simple : le cas où un utilisateur doit répondre à une question parmi une liste de réponses imposées comme o (oui) ou n (non). Si l'utilisateur répond autre chose (n'importe quoi), il faut lui reposer la question, jusqu'à ce qu'il réponde vraiment o ou n.

Pour créer une table de multiplication, de 3 par exemple, vous allez procéder comme si vous la récitiez :

- 3*1=3
- 3*2=6
- 3*3=9
- …
- 3*9=27
- 3*10=30

Vous allez donc multiplier 3 successivement par les nombres de 1 à 10. En algorithmique, vous connaissez les variables. Comment affecter une valeur de 1 à 10 successivement à une variable ? Avec une boucle !

Et si maintenant vous vouliez créer l'ensemble des tables de multiplication : tables de 1, de 2, etc, jusqu'à 10 ou plus ? Il vous faudra imbriquer deux boucles !

Si vous voulez calculer une puissance quelconque, une factorielle, sortir le plus grand des nombres parmi une liste de nombres saisis (en attendant les tableaux), etc : il faudra encore utiliser les boucles.

Dans le chapitre précédent, vous avez vu comment calculer les solutions d'un polynôme du second degré. Et si vous souhaitiez tracer sa courbe graphique via un programme (en PHP ou Java par exemple) ? Il vous faudra encore utiliser une boucle (et quelques astuces).

Vous verrez qu'avec les boucles, vous pourrez même extraire les racines carrées.

Ces exemples simples mettent en évidence au moins trois choses :

- Il existe plusieurs types de boucles : certaines ont un nombre fixe d'itérations, d'autres dépendent de

conditions de sortie que vous aurez à définir.

- Il est possible d'imbriquer plusieurs niveaux de boucles : vous pouvez faire des boucles dans des boucles, autant de fois que vous le voulez.

- Il existe une quantité infinie d'utilisation des boucles qui en font une structure incontournable en programmation pour la moindre des applications un tant soit peu complexe.

Tant Que

1. Structure générale

La boucle de type "Tant Que" permet la répétition d'un bloc d'instructions tant que la condition testée est vérifiée, donc vraie. Sa syntaxe est la suivante :

```
Tant Que booléen Faire
  Bloc d'instructions
FinTantQue
```

Lors de l'exécution du programme, celui-ci arrive sur l'instruction "Tant que". Il évalue l'expression booléenne (une ou plusieurs conditions, ou une seule variable). Si l'expression retourne VRAI, alors le programme exécute les instructions suivantes jusqu'à ce qu'il arrive au "FinTantQue". Arrivé ici, il remonte au "Tant Que" et évalue de nouveau l'expression booléenne, si c'est VRAI alors il exécute de nouveau les instructions et ainsi de suite, tant que l'expression retourne VRAI. Si l'expression devient fausse, alors le programme saute à l'instruction située juste après le "FinTantQue". Voici un simple exemple qui compte de 1 à 10 :

```
PROGRAMME TQUE1
VAR
  Cpt:entier
DEBUT
  Cpt←1
  Tant que Cpt<=10 Faire
    Afficher Cpt
    Cpt←Cpt+1
  FinTantQue
FIN
```

En PHP, la boucle "Tant Que" est représentée par "while()", avec l'expression booléenne entre parenthèses.

```
while(booleen) {
  /* bloc d'instructions */
}
```

Si une seule instruction est présente au sein de la boucle, les accolades sont inutiles.

```
while(booléen) instruction ;
```

Voici le code PHP correspondant à l'algorithme TQUE1 :

```
<html>
  <head><meta/>
    <title>tque1</title>
  </head>
  <body>
   <?php

  $cpt=1;
  while($cpt<=10) {
    echo "$cpt<br />";
    $cpt++;
  }
  ?>
  </body>
</html>
```

2. Boucles infinies et "break"

Faites toujours bien attention à ce que votre boucle dispose d'une condition de sortie. En effet rien ne vous empêche de faire des boucles infinies. Dans le cas d'une structure itérative "Tant Que", il suffit que la condition soit toujours VRAIE, par exemple :

```
Tant que VRAI Faire
  ...
FinTantQue
```

Ou encore,

```
Tant que a=a
  ...
FinTantQue
```

En PHP cela se traduit par :

```
...
while(true) {
...
}
```

Dans les deux cas, l'expression booléenne est toujours vraie donc le programme ne sort jamais de la boucle. La plupart des langages (C, C++, Java, PHP, etc) proposent des instructions spéciales qui permettent de sortir d'une boucle depuis n'importe quel endroit du bloc d'instructions (instruction **break**) ou même de relancer la boucle (remonter directement au Tant Que) sans exécuter le reste des instructions (instruction **continue**). L'une des premières choses que vous devez apprendre en algorithmique avec les boucles c'est que sans vouloir paraître excessif, l'utilisation de break et de l'instruction continue est **très déconseillée** : de nombreux programmeurs et pas forcément en herbe, parsèment l'intérieur de leurs boucles de conditions de sorties supplémentaires, qu'ils nomment souvent des "cas spéciaux" : "je n'ai pas prévu de gérer cela dans un cas général, alors je place une série d'exceptions…". Or l'objectif n'est pas de multiplier ces conditions, mais de toutes les réunir au sein de l'unique expression booléenne du Tant Que. Il faut donc réunir toutes les conditions d'arrêt de la boucle en un seul point.

Le "break" existe tel quel en PHP. Voici donc un exemple de ce que, théoriquement, il ne faut pas faire :

```
<html>
  <head><meta/>
    <title>Break</title>
  </head>
  <body>
  <?php

    $cpt=1;
    while(true) {
      echo "$cpt <br />";
      if($cpt==10) break;
      $cpt++;
    }
  ?>
  </body>
</html>
```

En algorithmique, il est toujours possible de trouver une expression booléenne, même si elle est longue et complexe, permettant d'éviter l'utilisation de "break" et de "continue". Si vous n'êtes pas d'accord, votre professeur risque de vous mettre au défi de trouver un exemple contradictoire. Peine perdue.

Il faut cependant modérer ces propos. L'interruption d'une boucle au milieu de celle-ci est déconseillée, certes. Mais comme pour tout, il faut se méfier des généralisations. Il y a évidemment des cas où il devient bien trop compliqué de créer des boucles uniquement pour respecter ce principe. S'il faut créer une expression booléenne à rallonge et bidouiller (c'est le mot, parfois) son bloc d'instructions avec des méthodes tarabiscotées (des drapeaux à tout va par exemple), d'autant plus que celui-ci prend déjà un grand nombre de lignes, autant utiliser un break. Au contraire, utiliser les breaks à tort et à travers n'est pas recommandable.

Le tout est de trouver un équilibre entre la condition de sortie et la lisibilité de l'algorithme.

Enfin, le but n'est pas de créer des boucles dont le bloc d'instructions fait dix pages (c'est une façon de parler). Dans ce cas, il est certes intelligent de réunir toutes les conditions de sortie en un point : cela améliore la lisibilité. Cependant, vous risquez de vous perdre dans votre propre programme (d'où les indentations). Vous apprendrez plus loin dans ce livre la notion de fonctions et de procédures qui permettront un découpage fin de vos blocs d'instructions.

3. Des exemples

a. Une table de multiplication

Pourquoi ne pas s'attaquer aux exemples cités ci-dessus et même plus, pour vous entraîner ? Commencez par la table de multiplication. Après avoir saisi le numéro de la table demandée, un compteur est initialisé à 1. Tant que ce compteur est inférieur ou égal à 10, on le multiplie par le numéro de table, puis après avoir affiché le résultat, on l'incrémente. Dans la dernière boucle, le compteur passe de 10 à 11. Une fois remonté au Tant Que, l'expression devient fausse : la boucle est terminée et le programme se termine.

```
PROGRAMME MUTLI1
VAR
  table,cpt,resultat:entiers
DEBUT
  Afficher "Quelle table de multiplication ?"
  Saisir table
  cpt←1
  Tant que cpt<=10 Faire
    resultat←cpt*table
    Afficher table,"x",cpt,"=",resultat
    cpt←cpt+1
  FinTantQue
FIN
```

Ce qui donne en PHP :

```
<html>
  <head><meta/>
    <title>Table de multiplication</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['table'])) {
  ?>
    <form method="GET">
      Table de multiplication : <input type="text"
size="4" name="table" />
      input type="submit" name="OK" />
    </form>
  <?php
  } else {
    $table=$_GET['table'];

    $cpt=1;
    while($cpt<=10) {
      $resultat=$table*$cpt;
      echo "${table}x${cpt}=$resultat <br />";
      $cpt++;
    }
  }
  ?>
  </body>
</html>
```

b. Une factorielle

Dans le même ordre d'idée, voici un petit algorithme qui calcule une factorielle. Pour rappel, la factorielle de n s'écrit n! et se calcule en multipliant toutes les valeurs de 1 à n. Ainsi 10!=10*9*8*7*6*5*4*3*2*1, soit 3628800. Dans chaque passage dans la boucle, il s'agit de multiplier le compteur par le résultat de la multiplication précédente. Notez qu'il est inutile de multiplier par un : cela ne change pas le résultat et le programme effectue une boucle de trop. De même, cette fois l'algorithme comptera à l'envers : il partira de n pour descendre jusqu'à deux.

```
PROGRAMME FACT
VAR
  cpt,resultat:entiers
DEBUT
  Afficher "Quelle factorielle ?"
  Saisir cpt
```

```
  resultat←cpt
  Tant que cpt>2 Faire
    cpt←cpt-1
    resultat←cpt*resultat
  FinTantQue
  Afficher resultat
FIN
```

Ce qui donne en PHP :

```
<html>
  <head><meta/>
    <title>Factorielle</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['cpt'])) {
  ?>
    <form method="GET">
      Factorielle de : <input type="text" size="4" name="cpt" />
      <input type="submit" name="OK" />
    </form>
  <?php
  } else {
    $cpt=$_GET['cpt'];
    $resultat=$cpt;

    while($cpt>2) {
      $cpt--;
      $resultat=$resultat*$cpt;
    }
    echo $resultat;
  }
  ?>
  </body>
</html>
```

c. x à la puissance y

Il s'agit cette fois d'élever une valeur à une puissance quelconque. Pour rappel, x^n est égal à x*x*x*x… n fois. C'est donc très simple : une boucle Tant Que qui va compter de 1 à n, dans laquelle le résultat de la multiplication précédente va être multiplié par x.

```
PROGRAMME puissance
VAR
  x,n,cpt,resultat:entiers
DEBUT
  Afficher "x,n ?"
  Saisir x,n
  cpt←1
  resultat←1
  Tant que cpt<=n Faire
    resultat←resultat*x
    cpt←cpt+1
  FinTantQue
  Afficher resultat
FIN
```

Cet algorithme fonctionne bien, mais ne gère pas tous les cas. En fait, il fonctionne uniquement si la puissance est supérieure ou égale à zéro. Si n vaut zéro, le programme ne rentre même pas dans la boucle et le résultat vaut 1. Si n vaut 1, resultat vaut x. Et pour les puissances négatives ? x^{-n} est égal à $1/x^n$.

Il s'agit donc de déterminer le signe de la puissance et de faire une division. Il faut aussi récupérer la valeur absolue de la puissance, ce que vous savez déjà faire. Dans l'algorithme suivant, un drapeau "signe" est utilisé pour savoir si n est négatif ou non, afin d'effectuer une division à la fin.

```
Variables x,n,signe,cpt,resultat en Numérique
Début
```

```
  Signe←0
  Ecrire "x,n ?"
  Lire x,n
  Si n<0 Alors
    Signe←1
    n←-n
  FinSI
  cpt←1
  resultat←1
  Tant que cpt<=n Faire
    resultat←resultat*x
    cpt←cpt+1
  FinTantQue
  Si signe=1 Alors
    Resultat←1/resultat
  FinSi
  Ecrire resultat
Fin
```

Ce qui donne en PHP :

```
<html>
  <head><meta/>
    <title>Factorielle</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['x'])) {
  ?>
    <form method="GET">
      x,y ? <input type="text" size="4" name="x"
/> <input type="text" size="4" name="y" />
      <input type="submit" name="OK" />
    </form>
  <?php
  } else {
    $x=$_GET['x'];
    $n=$_GET['y'];
    if($n<0) {

      $n=-$n;
      $signe=true;
    }
    $cpt=1;
    $resultat=1;

    while($cpt<=$n) {
      $resultat=$resultat*$x;
      $cpt++;
    }
    if($signe) $resultat=1/$resultat;
    echo $resultat;

  }
  ?>
  </body>
</html>
```

d. Toutes les tables de multiplication

Tout comme les tests, il est tout à fait possible d’imbriquer les boucles, c’est-à-dire de mettre une boucle dans une autre boucle, sur autant de niveaux que vous le souhaitez. Vous pouvez envisager 1, 2, 3, n niveaux, mais cela risque de devenir difficilement lisible. Voici un exemple à deux niveaux. Il s’agit tout simplement de calculer et d’afficher toutes les tables de multiplication de 1 à 10. Pour cela, deux boucles doivent être utilisées. La première va représenter la table à calculer, de 1 à 10. La seconde à l’intérieur de la première va multiplier la table donnée de 1 à 10. Vous avez donc deux boucles de 1 à 10. Simulez ce qu’il se passe :

- Première boucle, table des 1. Seconde boucle : exécution de 1*1, 1*2, 1*3 … 1*10

- Première boucle, table des 2. Seconde boucle : exécution de 2*1, 2*2, 2*3 … 2*10

- Première boucle, table des 3. Seconde boucle : exécution de 3*1, 3*2, 3*3 … 3*10

- …

- Première boucle, table des 10. Seconde boucle : exécution de 10*1, 10*2, 10*3 …

Voici l'algorithme correspondant. À chaque passage dans la première boucle, le compteur de la seconde boucle repasse à un. C'est dans la seconde boucle qu'a lieu le calcul et l'affichage de la table de multiplication.

```
PROGRAMME MULTI2
VAR
  table,cpt,resultat :entiers
DEBUT
  table←1
  Tant Que table<=10 Faire
    Afficher "Table des ",table
    cpt←1
    Tant que cpt<=10 Faire
      resultat←cpt*table
      Afficher table,"x",cpt,"=",resultat
      cpt←cpt+1
    FinTantQue
    Table←table+1
  FinTantQue
Fin
```

La même chose en PHP :

```
<html>
  <head><meta/>
    <title>Factorielle</title>
  </head>
  <body>
  <?php
    $table=1;
    while($table<=10) {
      echo "table des $table <br />";
      $cpt=1;
      while($cpt<=10) {
        $resultat=$table*$cpt;
        echo "${table}x${cpt}=$resultat<br />";
        $cpt++;
      }
      $table++;
    }
  ?>
  </body>
</html>
```

e. Saisie de notes et calcul de moyennes

Le but de l'algorithme suivant est d'inviter l'utilisateur à saisir des notes d'étudiants entre 0 et 20, de déterminer la note la plus basse, la note la plus haute et de calculer une moyenne. Tant que l'utilisateur n'a pas saisi la note négative -1, la saisie ne s'arrête pas. Les résultats sont ensuite affichés. Cet algorithme présente un intérêt certain : il faut contrôler la saisie des notes ; si l'utilisateur saisit autre chose qu'une note allant de -1 à 20, la question lui est reposée, sachant que -1 correspond à une fin de saisie. Procédez par étapes.

Commencez tout d'abord par cette partie de l'algorithme : la saisie des notes. Il faut pour cela utiliser une boucle dont l'unique condition de sortie est une note de -1.

```
PROGRAMME NOTE
VAR
```

```
  note:entier
Début
  Afficher "Entrez une note"
  Saisir note
  Tant que note>-1 Faire
    Si note >20 Alors
      Tant que note<-1 ou note>20 Faire
        Afficher "Erreur (0->20, -1 sortie) :"
        Saisir note
      FinTantQue
    FinSi
    Si note<>-1 Alors
      Afficher "Entrez une note"
      Saisir note
    FinSi
FinTantQue
Fin
```

Ce qui donne en PHP, depuis une console :

```
<?php

echo "Note (-1:sortie) ?";

$note=fgets(STDIN);
while($note>-1) {
  if($note>20) {
    while($note <=-1 || $note>20) {
      echo "Erreur ! Note (-1:sortie) ?";
      $note=fgets(STDIN);
    }
  }
  if($note>-1) {
    echo "Note actuelle :$note";
    echo "Note (-1:sortie) ?";
    $note=fgets(STDIN);
  }
}

?>
```

Cet algorithme montre qu'une simple répétition de saisie est plus complexe qu'il n'y paraît. Au tout début, l'utilisateur se voit poser une première fois la question, s'il répond "-1" tout de suite, il n'entre même pas dans la boucle. Dans la première boucle, une vérification est effectuée sur la validité de la note saisie : est-elle supérieure à 20 ? Si oui, alors il faut de nouveau effectuer une saisie et reposer en boucle la question autant de fois que nécessaire, c'est-à-dire tant que la note saisie n'est pas comprise entre -1 et 20. Sachant que -1 est une condition de sortie, aucun traitement, y compris la saisie d'une nouvelle note, ne doit être effectué dans ce cas. Vous avez ici une très belle application d'un cas où un "break" aurait pu effectivement être utilisé. Voici un exemple (incomplet) qui remplace le dernier "Si" de l'algorithme précédent :

```
...
Tant que note>-1 Faire
  ...
    Si note=-1 Alors
      break
    FinSi
    Ecrire "Entrez une note"
    Lire note
FinTantQue
```

La boucle principale peut en effet être directement interrompue dès qu'une note de -1 est rencontrée. Dans ce cas, il serait même possible d'aller plus loin en considérant la première boucle comme infinie et en effectuant dedans toutes les saisies et la condition de sortie :

```
Variable note en Numérique
Début
  Tant que VRAI Faire
    Ecrire "Entrez une note"
    Lire note
```

```
    Si note<-1 ou note>20 Alors
      Tant que note<-1 ou note>20 Faire
        Ecrire "Erreur (0->20, -1 sortie) :"
        Lire note
      FinTantQue
    FinSi
    Si note=-1 Alors
      break
    FinSi
  FinTantQue
Fin
```

Ce qui donne en PHP, via une console :

```
<?php
  while(true) {
    echo "Note (<-1:sortie) ?";
    $note=fgets(STDIN);
    if($note>20) {
      while($note<-1 || $note>20) {
        echo "Erreur ! Note (-1:sortie) ?";
        $note=fgets(STDIN);
      }
    }
    if($note==-1) break;
    echo "$note";
  }
?>
```

Il est encore possible d’aller plus loin en PHP avec l’instruction "continue". En effet plutôt que de créer une nouvelle boucle en cas d’erreur de saisie, pourquoi ne pas relancer celle-ci au début ?

Le programme devient donc :

```
<?php
  while(true) {
    echo "Note (<-1:sortie) ?";
    $note=fgets(STDIN);
    if($note>20) {
      echo "Erreur : nouvelle saisie...\n";
      continue;
    }
    if($note==-1) break;
    echo $note;
  }
?>
```

Comme vous pouvez le constater, cet algorithme est un peu plus court que l’original, avec notamment une saisie en moins, une expression booléenne en moins (dans la condition de la première boucle), un "sinon" en moins. Cependant fondamentalement cet algorithme est moins propre. Comme indiqué précédemment, le break est en principe déconseillé et il vaut mieux que la boucle elle-même ait une condition de sortie afin d’éviter une catastrophe en cas de modification du bloc d’instructions qu’elle contient. Il faut maintenant compléter l’algorithme d’origine pour y ajouter nos variables :

- Un compteur du nombre de notes.
- La note minimale.
- La note maximale.
- La somme de toutes les notes.
- La moyenne.

La moyenne s’effectue en sortie de la boucle principale, une fois l’ensemble des notes saisi. Il faut veiller à ce qu’au moins une note ait été saisie, sinon l’algorithme effectuerait une division par zéro, ce qui est évidemment impossible. Au sein de la boucle, et seulement si la note saisie est différente de -1, l’algorithme compare la plus

basse note avec la note actuelle, qui devient la plus basse le cas échéant, la même chose est faite pour la note la plus haute (vous prendrez soin d'initialiser correctement ces valeurs dès le début) et le total des notes est calculé. En sortie de boucle, il ne reste plus qu'à faire une division.

```
Variables note,cpt,min,max,sum,moy en Numérique
Début
  min←20
  max←0
  cpt←0
  sum←0
Ecrire "Entrez une note"
Lire note
Tant que note>-1 Faire
    Si note >20 Alors
      Tant que note<-1 ou note>20 Faire
        Ecrire "Erreur (0->20, -1 sortie) :"
        Lire note
      FinTantQue
    FinSi
    Si note<>-1 Alors
      cpt←cpt+1
      sum←sum+note
      Si note<min Alors
        min←note
      FinSI
      Si note>max Alors
        max←note
      FinSI
      Ecrire "Entrez une note"
      Lire note
    FinSi
FinTantQue
Si cpt>0 Alors
    moy←sum/cpt
    Ecrire "Nombre de notes : ",cpt
    Ecrire "Note la plus basse : ",min
    Ecrire "Note la plus haute : ",max
    Ecrire "Moyenne : ",moy
Sinon
    Ecrire "Aucune note n'a été saisie."
FinSI
Fin
```

Le programme suivant, en PHP, est une adaptation du programme précédent. Vous n'y voyez pas de boucle, ou plutôt elle n'est pas explicite : vous bouclez en validant la saisie d'une note par le bouton **OK**. La difficulté, en PHP et HTML, est de conserver les valeurs d'une page à l'autre. Il existe diverses méthodes. Celle présentée dans cet exemple consiste à conserver les valeurs dans des champs cachés du formulaire.

```
<html>
  <head><meta/>
    <title>moyenne max 3</title>
  </head>
  <body>
  <?php
  $min=20;
  $max=0;
  $cpt=0;
  $sum=0;
  $note=0;
  if(isset($_GET['note'])) {
    $note=$_GET['note'];
    $cpt=$_GET['cpt'];
    $min=$_GET['min'];
    $max=$_GET['max'];
    $sum=$_GET['sum'];
    $cpt++;
  }

  if($note>20) {
```

```
      echo "Erreur : nouvelle saisie...";
      $cpt--;
    }
    else if($note!=-1) {
      $sum=$sum+$note;
      if($note<$min) $min=$note;
      if($note>$max) $max=$note;

      echo $note;
      if($cpt!=0) {
        $moy=$sum/$cpt;

        echo "cpt:$cpt";
        echo "min:$min";
        echo "max:$max";
        echo "moy:$moy";
      }
    }
    ?>
      <form method="GET">
      note (-1 = sortie) ? : <input type="text" size="4" name="note" />
        <input type="hidden" name="cpt" value="<?=$cpt ?>" />
        <input type="hidden" name="sum" value="<?=$sum ?>" />
        <input type="hidden" name="min" value="<?=$min ?>" />
        <input type="hidden" name="max" value="<?=$max ?>" />

        <input type="submit" name="OK" />
      </form>

    </body>
  </html>
```

f. Rendez la monnaie

Le but de cet algorithme est de calculer un rendu de monnaie en fonction de la valeur des pièces et des billets, en déterminant combien de billets de telle valeur ou de pièces de telle valeur il faut rendre. Il s'agit donc de transformer une somme en coupures correspondantes. Par exemple 1898,67 euros peuvent se décomposer en :

- 3 billets de 500 euros
- 1 billet de 200 euros
- 1 billet de 100 euros
- 1 billet de 50 euros
- 2 billets de 20 euros
- 1 billet de 5 euros
- 1 pièce de 2 euros
- 1 pièce de 1 euro
- 1 pièce de 50 centimes
- 1 pièce de 10 centimes
- 1 pièce de 5 centimes
- 1 pièce de 2 centimes

Autant le dire tout de suite, cet algorithme est le paradis des boucles. Il en faudra une par valeur faciale de billet ou de pièce. Le principe est en effet très simple. Il s'agit de soustraire au sein d'une boucle la valeur faciale du billet, par exemple 500 euros, au montant total, tant que ce montant est supérieur ou égal à la valeur du billet. À chaque passage dans la boucle, on compte un billet. Puis on passe au billet suivant et ainsi de suite. Voici un exemple pour 1700 euros avec uniquement des billets de 500 euros.

```
Variables montant, nb500 en Numérique
Début
  montant←1700
  nb500←0
  Tant Que montant>=500 Faire
    nb500←nb500+1
    montant←montant-500
  FinTantQue
  Ecrire nb500, montant
Fin
```

Ce qui donne en PHP :

```
<html>
  <head><meta/>
    <title>Factorielle 3</title>
  </head>
  <body>
  <?php

    $montant=1700;
    $nb500=0;

    while($montant>=500) {
      $nb500++;
      $montant=$montant-500;
    }
    echo "Nombre de billets de 500: $nb500 <br />";
    echo "Reste : $montant";
  ?>
  </body>
</html>
```

Que se passe-t-il à la sortie de la boucle ? nb500 vaut 3 et montant vaut 200 : c'est le reste. Aussi, il faut généraliser l'algorithme pour toutes les valeurs.

```
Variables montant,nb500,nb200,nb100,nb50,nb20,nb10,nb5 en Numérique
Variables nb2,nb1,nb05,nb02,nb01,nb005,nb002,nb001 en Numérique
Début
  montant←1700
  nb500←0
  nb200=0
  nb100=0
  nb50=0
  nb20=0
  nb10=0
  ...
  nb002=0
  nb001=0
  Tant Que montant>=500 Faire
    nb500←nb500+1
    montant←montant-500
  FinTantQue
  Tant Que montant>=200 Faire
    nb200←nb200+1
    montant←montant-200
  FinTantQue
  Tant Que montant>=100 Faire
    nb100←nb100+1
    montant←montant-100
  FinTantQue
  Tant Que montant>=50 Faire
    nb50←nb50+1
```

```
    montant←montant-50
  FinTantQue
  Tant Que montant>=20 Faire
    nb20←nb20+1
    montant←montant-20
  FinTantQue
  ...
  Tant Que montant>=0.02 Faire
    nb002←nb002+1
    montant←montant-0.02
  FinTantQue
  Tant Que montant>=0.01 Faire
    nb001←nb001+1
    montant←montant-0.01
  FinTantQue
  Si nb500>0 Alors
    Ecrire nb500," billets de 500 euros"
  FinSI
  Si nb200>0 Alors
    Ecrire nb200," billets de 200 euros"
  FinSI
  ...
  Si nb002>0 Alors
    Ecrire  nb002," pièces de 2 centimes"
  FinSI
  Si nb001>0 Alors
    Ecrire nb001," pièces de 1 centime"
  FinSI
Fin
```

Cet algorithme est désespérant. Il est parfait en terme de fonctionnement et de logique, mais il est épouvantablement long, proportionnel au nombre de coupures disponibles dans chaque pays. Tellement long d'ailleurs que certaines parties totalement évidentes ont été remplacées par des "...". Est-il possible de faire plus court ? Il est évidemment possible de faire plus court. Vous verrez comment faire au prochain chapitre avec les tableaux, puis dans le suivant encore avec les fonctions.

g. Trois boucles

Un dernier exemple de boucle "Tant Que" va vous montrer une intégration avec trois boucles. Le but de cet anodin algorithme est de trouver pour quelles valeurs de A, B et C, $ABC=A^3+B^3+C^3$, A représentant les centaines, B les dizaines et C les unités. La recherche sera limitée pour chaque valeur entière comprise entre 1 et 10 (bien entendu, vous pouvez augmenter l'intervalle mais celui-ci n'a pas été choisi au hasard). L'algorithme nécessite trois boucles pour chacune des valeurs. C'est bien entendu au sein de la dernière boucle que les valeurs sont calculées et les résultats affichés en cas d'égalité.

```
Variables a,b,c,nb1,nb2 en Numérique
Début
  a←1
  b←1
  c←1
  Tant que a<=10 Faire
    Tant que b<=10 Faire
      Tant que c<=10 Faire
        nb1←a*100+b*10+c
        nb2←a³+b³+c³
        Si nb1=nb2 Alors
          Ecrire nb1,a,b,c
        FinSI
        c←c+1
      FinTantQue
      b←b+1
    FinTantQue
    a←a+1
  FinTantQue
Fin
```

Si vous convertissez ce programme en PHP, vous devriez obtenir seulement deux possibilités : 153 et 371. En effet $3^3+7^3+1^3=27+343+1=371$...

Voici le programme équivalent en PHP :

```
<html>
  <head><meta/>
    <title>Trois boucles</title>
  </head>
  <body>
  <?php
    $x=1;
    $y=1;
    $z=1;
    while($x<=10) {
      while($y<=10) {
        while($z<=10) {
          $nb1=$x*100+$y*10+$z;
          $nb2=pow($x,3)+pow($y,3)+pow($z,3);
          if($nb1==$nb2) {
            echo "$nb1=$nb2, $x $y $z <br />";
          }
          $z++;
        }
        $y++;
      }
      $x++;
    }
  ?>
  </body>
</html>
```

Répéter ... Jusqu'à

1. Différences fondamentales

Malgré l'abondance d'exemples vus jusqu'à présent, la structure itérative "Tant Que" n'est pas la seule. Même s'il est possible de tout programmer avec ce type de boucle, il en manque encore deux, dont la structure "Répéter ... Jusqu'à". Son pseudo-code est le suivant :

```
Répéter
  Bloc d'instructions
Jusqu'à booléen
```

Le "Répéter" ressemble fortement au "Tant que" avec cependant deux importantes différences :

- Quoi qu'il arrive, il y aura toujours au moins un passage dans la boucle : le bloc d'instructions sera exécuté au moins une fois,
- L'expression booléenne finale est inversée. Un " tant que a !=1 " devient un "jusqu'à a=1".

Le "jusqu'à" se comprend comme "jusqu'à ce que la condition soit vérifiée". Pour faire une boucle infinie, il faut donc faire :

```
Répéter
  Bloc d'instructions
Jusqu'à FAUX
```

Pour reprendre l'algorithme de saisie du relevé de notes qui avait posé quelques problèmes, celui-ci devient un peu plus simple :

```
PROGRAMME REPETE
VAR
  Note:entier
Début
  Répéter
    Ecrire "Saisissez une note"
    Lire note
    Si note<-1 OU note>20 Alors
      Répéter
        Ecrire "Erreur (0->20, -1 sortie) :"
        Lire note
      Jusqu'à note>=-1 ET note <=20
    FinSI
    Si note!=-1 Alors
      ...
    FinSI
  Jusqu'à note=-1
Fin
```

Le langage PHP propose une structure "répéter ... Tant Que". Il suffit donc uniquement d'inverser la condition de sortie du "jusqu'à".

```
do {
  /* bloc d'instructions */
}while(condition) ;
```

Le programme PHP avec quelques facilités pourrait être :

```
<?php

  do{
    echo "Note (<-1:sortie) ?";
    $note=fgets(STDIN);
    if($note>20) {
```

```
      echo "Erreur : nouvelle saisie...";
      continue;
    }
    if($note>-1) echo $note;
  }while($note>-1);
?>
```

Notez bien attentivement la condition de sortie centrale : la note doit être supérieure ou égale à -1 ET inférieure ou égale à 20 pour sortir de la saisie. Avec le "Tant que", la boucle continuait tant que la note était inférieure à -1 OU supérieure à 20. Une nuance, mais de taille.

Ces nuances sont l'une des raisons qui font que des étudiants et des programmeurs, pourtant parfois chevronnés, s'y perdent. C'est aussi l'une des raisons qui fait que la boucle "Répéter … jusqu'à" n'est pas présente dans certains langages comme le C ou Java. En Java ou en PHP cependant, vous trouvez une boucle équivalente au "Répéter … Tant Que", le "do … while", qui reprend le fait d'une itération obligatoire, mais avec les mêmes expressions booléennes que le "Tant Que" initial.

2. Quelques exemples adaptés

a. La factorielle

Inutile de décrire à nouveau le principe. La boucle doit être quittée quand le compteur vaut 2.

```
Variables cpt, resultat en Numérique
Début
  Ecrire "Quelle factorielle ?"
  Lire cpt
  resultat←cpt
  Répéter
    cpt←cpt-1
    resultat←cpt*resultat
  Jusqu'à cpt=2
  Ecrire resultat
Fin
```

b. Les trois boucles

Là encore, même programme mais style différent. Notez que ici les compteurs partent de 0 afin d'être incrémentés en début de boucle, dans le but de rendre la condition de sortie plus lisible (égale à 10).

```
Variables a,b,c,nb1,nb2 en Numérique
Début
  a←0
  b←0
  c←0
  Répéter
    a←a+1
    Répéter
      b←b+1
      Répéter
        c←c+1
        nb1←a*100+b*10+c
        nb2←a³+b³+c³
        Si nb1=nb2 Alors
          Ecrire nb1,a,b,c
        FinSI
      Jusqu'à c=10
    Jusqu'à b=10
  Jusqu'à a=10
Fin
```

En PHP :

```
<html>
  <head><meta/>
```

```
    <title>Trois boucles 2</title>
  </head>
  <body>
  <?php
    $x=1;
    do {
      $y=1;
      do {
        $z=1;
        do {
          $nb1=$x*100+$y*10+$z;
          $nb2=pow($x,3)+pow($y,3)+pow($z,3);
          if($nb1==$nb2) {
            echo "$nb1=$nb2, $x $y $z <br />";
          }
          $z++;
        }while($z<=10);
        $y++;
      }while($y<=10);
      $x++;
    }while($x<=10);
  ?>
  </body>
</html>
```

Pour ... Fin Pour

1. Une structure pour compter...

Troisième et dernière structure itérative de l'algorithmique, le **"Pour ... Fin Pour"** est une boucle à l'usage quasi-exclusif des compteurs. À chaque passage dans la boucle, un compteur est incrémenté ou décrémenté, selon le cas. On dit alors qu'il s'agit d'une **structure incrémentale**.

Sa syntaxe en pseudo-code est la suivante :

```
Pour variable De début à fin [PAS pas] Faire
  Bloc d'instructions
Fin Pour
```

À chaque passage dans la boucle, la variable prendra successivement chacune des valeurs dans l'intervalle [a;b] (a et b inclus). Le pas est optionnel et est de 1 par défaut. Le pseudo-code suivant compte de 1 à 10 :

```
Variable cpt en Numérique
Début
  Pour cpt De 1 à 10 Faire
    Ecrire cpt
  Fin Pour
Fin
```

Il est possible de trouver des syntaxes légèrement différentes, elles sont cependant toutes équivalentes :

```
Pour variable Allant De début à fin [PAS pas] Faire
  Bloc d'instructions
Fin Pour
```

Ou encore :

```
Pour compteur ← début à fin [Pas pas]
  Bloc d'instructions
compteur suivant
```

Dans cette dernière forme, il est intéressant de constater qu'il est plus simple, dans le cas de boucles contenant un gros bloc d'instructions, de s'y retrouver, la variable étant répétée dans la syntaxe de fin de boucle. Enfin, il est possible de trouver des syntaxes alternatives dérivant de ces trois dernières.

2. ... mais pas indispensable

Vous aurez rapidement compris que cette boucle ne sert que pour des compteurs. Autrement dit, tout ce qu'elle propose est déjà intégralement possible avec les structures "Tant Que" et "Répéter". Simplement avec le "Pour" vous n'avez pas à faire vous-même le calcul du compteur. C'est donc une simplification.



Il n'existe aucun cas où la structure "Pour ... Fin Pour" est strictement nécessaire. Elle ne fait que simplifier les autres structures itératives lors de l'utilisation de compteurs.

3. Quelle structure choisir ?

Mais alors quand utiliser telle ou telle structure ? On emploie une structure "Pour" lorsqu'on connaît à l'avance le nombre d'itérations nécessaires au traitement. Ce nombre peut être fixe ou calculé par avance avant la boucle, peu importe. La boucle "Pour" est déterministe : son nombre d'itérations est fixé une fois pour toutes et est en principe invariable (bien qu'il reste possible de tricher).

On emploie les structures "Tant Que" ou "Répéter" lorsqu'on ne connaît pas forcément à l'avance le nombre d'itérations qui seront nécessaires à l'obtention du résultat souhaité. L'exemple le plus concret est représenté par la saisie des notes : on peut en saisir une, 200, aucune, mais on le sait pas à l'avance. Mais cela peut être aussi la lecture des lignes d'un fichier (on n'en connaît pas le nombre par avance), d'enregistrements dans une base de

données, un calcul complexe devant retourner une précision particulière, un nombre de tours dans un jeu (le premier qui gagne sort de la boucle), etc.

4. Un piège à éviter

Tout comme il faut éviter les boucles infinies (sauf si c'est parfaitement volontaire), il faut aussi éviter quelques erreurs qui peuvent se révéler très surprenantes selon le cas (rappelez-vous la Loi de Murphy). Voici un exemple de ce qu'il ne faut pas faire :

```
Variable x en Numérique
Début
  Pour x allant de 1 à 31 Faire
    Ecrire x
    x←x*2
  Fin Pour
Fin
```

L'erreur (sauf si c'est ce que vous vouliez explicitement, ce qui s'appelle jouer avec le feu) est de modifier vous-même le compteur utilisé par la boucle au sein même de cette boucle. Que se passe-t-il dans cet exemple ? Vous passez de 1 à 2, puis … Quoi ? 3 ? 6 ? 7 ? 14 ? Ca devient n'importe quoi et aucun nombre d'itérations ne peut être prévu. Vous ne savez plus quand vous allez sortir de la boucle, ni même la nouvelle valeur du compteur à chaque itération. Selon les langages, vous aurez au mieux un fonctionnement à peu près conforme à ce que vous attendiez, au pire du grand n'importe quoi. Or en informatique, sauf à travailler pour un éditeur peu scrupuleux sur la qualité, un "à peu près" est inacceptable.



Ne modifiez jamais un compteur de boucle "Pour" au sein de celle-ci. Si cela s'avère vraiment nécessaire, modifiez votre boucle pour utiliser une structure "Tant Que" ou "Répéter".

5. Quelques exemples

a. De nouveau trois boucles

Promis à devenir un grand classique, puisque c'est la troisième fois que vous le voyez sous trois formes différentes donc, voici l'exemple des trois boucles, preuve s'il en est que le "Pour … Fin Pour" est bien pratique mais totalement optionnel. L'intérêt est évidemment ici de produire un code plus succinct et encore plus lisible.

```
Variables a,b,c,nb1,nb2 en Numérique
Début
  Pour a de 1 à 10 Faire
    Pour b de 1 à 10 Faire
      Pour c de 1 à 10 Faire
        nb1←a*100+b*10+c
        nb2←a³+b³+c³
        Si nb1=nb2 Alors
          Ecrire nb1,a,b,c
        FinSI
      Fin Pour
    Fin Pour
  Fin Pour
Fin
```

Ce qui en PHP donne :

```
<html>
  <head><meta/>
    <title>Trois boucles</title>
  </head>
  <body>
  <?php
    for($x=1;$x<=10;$x++) {
      for($y=1;$y<=10;$y++) {
        for($z=1;$z<=10;$z++) {
```

```
            $nb1=$x*100+$y*10+$z;
            $nb2=pow($x,3)+pow($y,3)+pow($z,3);
            if($nb1==$nb2) {
              echo "$nb1=$nb2, $x $y $z <br />";
            }
          }
        }
      }
    ?>
    </body>
</html>
```

b. La factorielle

Encore un classique : avec une factorielle de n, vous savez à l'avance le nombre d'itérations nécessaire : n. C'est donc une application de choix pour la structure "Pour ... Fin Pour".

```
Variables cpt, i, resultat en Numérique
Début
  Ecrire "Quelle factorielle ?"
  Lire cpt
  resultat←1
  Pour i de 2 à cpt
    resultat←i*resultat
  Fin Pour
  Ecrire resultat
Fin
```

En PHP :

```
<html>
  <head><meta/>
    <title>Factorielle 3</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['cpt'])) {
  ?>
    <form method="GET">
      Factorielle de : <input type="text" size="4" name="cpt" />
      <input type="submit" name="OK" />
    </form>
  <?php
  } else {
    $cpt=$_GET['cpt'];
    $resultat=1;

    for($i=2;$i<=$cpt;$i++) $resultat=$resultat*$i;


    echo $resultat;
  }
  ?>
  </body>
</html>
```

c. Racine carrée avec précision

Voici une innovation : le calcul d'une racine carrée. Savez-vous extraire une racine carrée à la main ? Avant l'apparition des calculatrices, les lycéens et étudiants utilisaient soit des tables ou des règles à calcul, soit les calculaient eux-mêmes. Il serait possible de décrire la méthode utilisée mais il manque encore quelques notions. Une méthode plus mathématique est l'algorithme de Héron d'Alexandrie. Cet homme aussi appelé Héron l'Ancien était un mathématicien, mécanicien et ingénieur grec né à Alexandrie au 1er siècle de notre ère. Il a écrit de nombreux traités et laissé quelques formules dont l'une permettant de calculer l'aire d'un triangle en fonction de la longueur de ses côtés et une autre permettant d'approcher la valeur d'une racine carrée de manière récursive. Voici comment trouver la formule :

$$x=\sqrt{a}$$

$$donc\ x^2=a$$

$$donc\ 2x^2=x^2+a$$

$$donc\ 2x=\frac{x^2+a}{x}$$

$$donc\ x=\frac{x^2+a}{2x}$$

$$donc\ x=\frac{x(x+\frac{a}{x})}{2x}$$

$$donc\ x=\frac{x+\frac{a}{x}}{2}$$

$$on\ obtient\ enfin\ la\ suite\ x_{i+1}=\frac{1}{2}(x_i+\frac{a}{x_i})$$

Formule de Héron d'Alexandrie

La suite finale permet de calculer la racine carrée en fonction d'une valeur initiale arbitraire x_0. En principe, on utilise la valeur entière approchée de la racine carrée, la formule permettant initialement d'obtenir rapidement les nombres situés après la virgule. Donc si vous cherchez la racine carrée de 40, sachant que 6x6 vaut 36 et que 7*7 vaut 49, la racine carrée est comprise entre les deux, vous devriez mettre 6. Cependant dans la pratique, n'importe quelle valeur peut convenir, avec un nombre important d'itérations vous obtiendriez toujours le résultat attendu, autant en plaçant 1 que 10000.

Il est cependant très intéressant de pouvoir optimiser le calcul en récupérant l'entier le plus proche. L'algorithme sera extrêmement précis. Celui-ci contiendra deux boucles. La première de type "Tant Que", chargée de calculer l'entier x le plus proche de la racine carrée de a. La seconde de type "Pour" calculera les décimales. Le nombre d'itérations permettra de déterminer une précision. D'ailleurs, inutile d'exécuter cette seconde boucle si le résultat de la première correspond à la racine recherchée.

```
Variables i,x,a,cpt en Numérique
Début
  x←1
  α←39
  cpt←5
  Tant Que (x*x)<a Faire
    x←x+1
  Fin Tant Que
  Si (x*x) !=a Alors
    x←x-1
    Pour i de 1 à cpt Faire
      x←0.5*(x+a/x)
    Fin Pour
  FinSi
  Ecrire "La racine de ",a," est ",x
Fin
```

Vous allez être étonné de la pertinence et de la précision des résultats. Ainsi en seulement trois ou quatre itérations, la précision est suffisante pour la plupart des applications. Malheureusement, vous le verrez dans la suite de l'ouvrage, c'est beaucoup de travail pour pas grand chose : les langages sont fournis avec des instructions particulières permettant d'effectuer ces calculs, d'autant plus que les fameux FPU (coprocesseurs arithmétiques) disposent d'une instruction en dur rien que pour cela (FSQRT par exemple sur un vieux Motorola MC68881).

En attendant, voici la transcription de ce calcul avancé en PHP :

```
<html>
```

```
  <head><meta/>
    <title>racine</title>
  </head>
  <body>
  <?php
    $x=1;
    $a=31;
    $cpt=5;
    while($x*$x<$a) {
      $x++;
    }
    if($x*$x!=$a) {
      $x--;
      for($i=1;$i<=$cpt;$i++) {
        $x=0.5*($x+($a/$x));
      }
    }
    echo "$x<br />";
    echo pow($x,2)."<br />";
  ?>
  </body>
</html>
```

d. Calcul du nombre PI

Sachant maintenant à l'aide des boucles à peu près tout faire, notamment calculer une racine carrée, il serait intéressant de trouver une application encore plus précise. Et pourquoi pas tenter d'approcher la valeur de PI ? Il existe plusieurs moyens d'approcher cette valeur. PI est la circonférence d'un cercle dont le diamètre est 1. Sans vous exposer ici les détails de la méthode, sachez que Leonhard Euler, grand savant du XVIIIème siècle, a résolu un problème connu de longue date : la détermination de la somme des inverses des carrés d'entier. La formule est la suivante :

$$\frac{\pi^2}{6}=\frac{1}{1^2}+\frac{1}{2^2}+\frac{1}{3^2}+\frac{1}{4^2}+\frac{1}{5^2}+\frac{1}{6^2}...$$

Remarquez la présence d'une itération sur les puissances de chaque dénominateur (diviseur). Voici une application de la boucle "Pour". De même, remarquez que PI est élevé au carré. Il faudra donc que vous effectuiez une racine carrée à la fin pour obtenir le bon résultat. Donc :

- Effectuer n divisions successives de $1/n^2$.
- Multiplier ce résultat par 6.
- Extraire la racine carrée de ce résultat pour obtenir PI.

```
Variables i,x,a,cpt en Numérique
Début
  cpt←100000
  a←2
  Pour i de 2 à cpt Faire
    a←a+1/(i*i)
  Fin Pour
  a←a*6
  x←1
  Tant Que (x*x)<a Faire
    x←x+1
  Fin Tant Que
  Si (x*x) !=a Alors
    x←x-1
    cpt←10
    Pour i de 1 à cpt Faire
      x←0.5*(x+a/x)
    Fin Pour
  FinSi
  Ecrire "La valeur de PI est ",x
Fin
```

Question : Combien faut-il d'itérations pour obtenir sept chiffres corrects après la virgule ? Voici le programme PHP correspondant, qui devrait vous fournir une petite idée du résultat :

```
<html>
  <head><meta/>
    <title>PI</title>
  </head>
  <body>
  <?php
    $cpt=100000000;
    $a=1;
    for($i=2;$i<=$cpt;$i++) {
      $a=$a+1/($i*$i);
    }
    $a=$a*6;
    $x=1;
    $cpt=10;
    while($x*$x<$a) {
      x++;
    }
    if($x*$x!=$a) {
      $x--;
      for($i=1;$i<=$cpt;$i++) {
        $x=0.5*($x+($a/$x));
      }
    }
    echo $x;
  ?>
  </body>
</html>
```

Présentation

1. Principe et définition

a. Simplifier les variables

Jusqu'à présent, les types de données que vous avez rencontrés sont des scalaires, sauf pour les chaînes de caractères. Pour rappel un scalaire est un type de donnée qui ne représente qu'une seule variable à la fois. Un entier, un caractère, un réel, un booléen, etc, sont des scalaires. Une chaîne de caractères non : il s'agit d'une suite, ou liste, de caractères, les uns après les autres. Une chaîne est donc une liste ordonnée par vos soins de scalaires. Les langages proposent souvent un type pour les chaînes de caractères, mais c'est une facilité offerte par ceux-ci. Un langage comme le C n'en propose pas. En algorithmique, vous utilisez le type "Alphanumérique", il vous faudra alors faire attention lors de la conversion en C.

Mais alors comment se représenter une chaîne de caractères avec un type scalaire ? Il faut pour cela se rappeler comment sont placées en mémoire les informations. La mémoire de l'ordinateur est composée de cases pouvant contenir certaines informations. Ces cases sont numérotées (on parle d'adresse de la case) et contiennent des données. Ces données représentent ce que vous voulez selon le contexte de leur utilisation. Vous pouvez par exemple partir du principe qu'elles contiennent des scalaires. Si une case mémoire contient le nombre 65, ce peut être la valeur entière 65 ou encore le code ASCII du caractère "A". Une case mémoire peut parfaitement contenir l'adresse d'une autre case mémoire : c'est un peu plus compliqué que cela en a l'air et ce sera l'objet d'un plus long exposé dans la suite de cet ouvrage.

Une chaîne de caractères est donc une suite de scalaires de type Caractère, les uns derrière les autres dans des cases mémoires en principes contiguës. Selon le même principe, si vous reprenez l'exemple de la saisie des notes d'étudiants, ne pensez-vous pas qu'il serait plus pratique de pouvoir conserver ces notes pour la suite du programme ? Il serait alors possible de les réutiliser à volonté pour de nouveaux calculs, voire même pour les sauver dans un fichier, les imprimer, les consulter, etc.

Jusqu'à présent, le seul moyen dont vous disposiez était de faire une boucle de saisie de notes et dedans, de tenter de faire les calculs au fur et à mesure. L'autre possibilité était de poser n fois la même question et de placer les résultats dans n variables différentes. Imaginez ceci :

```
...
Lire N1
Lire N2
...
Lire N20
Moy←(N1+N2+...+N20)/20
...
```

Ridicule, n'est-ce pas ? Maintenant, si vous savez qu'il y a vingt élèves dans une classe et donc vingt notes à saisir, ne serait-il pas plus simple de remplacer toutes les variables par une seule, mais qui pourrait contenir toutes les notes ? L'idée serait donc d'avoir un nom de variable mais qui pourrait associer une note à un numéro. Prenez la variable "note". Il suffirait alors de dire que "note 1 vaut 15, note 2 vaut 17, note 3 vaut 8, etc.".

Un ensemble de valeurs représenté par le même nom de variable et où chaque valeur est identifiée par un numéro s'appelle un tableau. Le numéro qui sert à identifier un élément (une valeur) du tableau s'appelle un indice. En représentation algorithmique, un élément du tableau est représenté par le nom de la variable auquel on accole l'indice entre crochets :

```
Note[1]←15
```



Un tableau n'est pas un type de données, mais une liste d'éléments d'un type donné. On parlera d'un tableau de n éléments de type numérique, ou alphanumérique, etc.

---

b. Les dimensions

Faites courir un peu plus votre imagination : maintenant vous avez trois classes de vingt élèves. Devez-vous utiliser trois tableaux ? Ce qu'il y a de bien avec les tableaux, c'est que l'on peut ajouter des indices aux indices. C'est très facile à appréhender avec deux indices.

```
Note[1][10]←17
```

Ceci pourrait (au conditionnel car soumis aux contraintes de la numérotation des éléments) se traduire par : la 10ème note de la 1ère classe.

Ajouter un indice à un tableau s'appelle ajouter une dimension à un tableau. Avec une dimension le tableau peut être représenté sur une ligne. Avec deux dimensions, le tableau peut être représenté en lignes et colonnes, comme dans un tableur ou une grille quelconque. Et avec trois dimensions ? Sous forme de cube, avec un axe de profondeur. Au-delà, c'est plus difficile à représenter, aussi il faut parfois utiliser des analogies avec des choses de la vie courante. Ainsi pour trois dimensions, imaginez un grand casier avec x cases en largeur, y cases en hauteur et dont chaque tiroir est décomposé en z petites cases. Ces représentations restent totalement virtuelles, une vue de l'esprit. De nombreux tableaux n'ont absolument pas pour but de représenter des lignes et des colonnes. Un tableau à deux dimensions peut parfaitement représenter un jeu de morpion, une matrice, une classe et les notes des étudiants associées...

Le piège avec les tableaux à plusieurs dimensions, c'est la taille qu'ils occupent en mémoire. Imaginez dix écoles, disposant de dix classes chacune de vingt élèves. Nous voulons placer les notes dans un tableau. Voici donc un tableau de trois dimensions :

```
Note[1..10][1..10][1..20]
```

Combien de notes pourra obtenir le tableau ? 10x10x20 : 2000 notes ! Si l'élément fait un octet, vous approchez les 2 Ko. Mais si l'élément du tableau contient un réel sur 64 bits, c'est 16 Ko qui sont utilisés ! Pourtant les indices semblaient si peu élevés.

c. Les types

Un tableau n'est pas un type de données mais un ensemble de valeurs, elles-mêmes typées, regroupées et indicées sous un nom de variable unique. Pouvez-vous créer un tableau contenant n'importe quel type de valeurs ? Attention à l'interprétation de cette question. Un tableau contient-il n valeurs du même type, ou au contraire n valeurs de types différents ?

En algorithmique, le principe est simple : un tableau contient n éléments de même type. Autrement dit, vous allez déclarer un tableau de vingt notes en numérique, dix réels, cinq chaînes de caractères, etc.

Cependant en dehors du pseudo-code algorithmique la définition, la déclaration et l'utilisation des tableaux dépendent fortement du langage. Les tableaux simples en Java ou en C par exemple ne contiennent qu'un seul type possible de valeurs. Tandis qu'en PHP, vous pouvez mélanger tout ce que voulez, l'indice 1 contenant un entier, l'indice 2 du texte, etc.

Cela peut être un peu déroutant à l'usage, mais ces langages, souvent appelés non typés (c'est discutable) sont d'une souplesse incomparable.

En attendant, respectez en algorithmique le fait qu'un tableau a en principe un nombre d'indices fini et qu'ils sont typés une fois pour toute.

d. Déclaration

En pseudo-code algorithmique, les tableaux se déclarent au même endroit que les variables, juste avant le début du traitement lui-même, sous cette forme :

```
VAR
  MonTableau:tableau[1..nbelements] d'entiers
  MonTab2:tableau[1..dim1][1..dim2] de réels
```

Entre les crochets, placez le nombre d'éléments du tableau.

Il est possible d'initialiser le contenu du tableau à sa création comme ceci :

```
VAR
  mois :tableau[1..12]<-{"janvier",...,"décembre"} de chaînes
```

Ce même tableau pourrait être placé dans la section CONST, ce qui en ferait une constante.

Comme cela sera revu un peu plus loin, les indices des tableaux peuvent démarrer à 0 ou 1, selon les langages, les usages, les professeurs, etc. Il n'y a malheureusement pas de règle précise en ce domaine. L'évidence, vis-à-vis de l'organisation de la mémoire de l'ordinateur, voudrait que la numérotation démarre à zéro : cela simplifie les calculs de la position des différents éléments du tableau dans la mémoire. Cependant comment alors comprendre ce tableau ?

```
Valeurs:tableau[1..10] de réels
```

Selon les usages, si ce tableau représente dix valeurs alors :

- Si la numérotation commence à 1, les indices vont de valeur[1] à valeur[10].
- Si la numérotation commence à 0, les indices vont de valeur[0] à valeur[9]. C'est le cas du langage C, du Java ou du PHP.

Certaines notations algorithmiques sont encore plus surprenantes (pour ne pas être méchant) : la valeur indiquée entre crochets peut correspondre au nombre maximal d'indices en partant de zéro. C'est ainsi que le tableau valeurs contiendrait onze éléments ! Ce ne sera pas le cas ici.

Dans la suite, les indices commenceront à 1 pour aller jusqu'à n, n étant le nombre d'éléments du tableau. Le tableau valeurs[1..10] aura donc bien dix éléments, numérotés de 1 à 10. Comme il ne s'agit pas d'une règle absolue dans tous les langages vous prendrez bien soin de vérifier ce qu'il en est lorsque vous écrirez vos programmes. Si vous êtes étudiant, suivez la représentation fournie par vos professeurs, éventuellement précisez les règles que vous appliquez aux indices. Dans tous les cas, n'accusez pas l'auteur de cet ouvrage !

e. Utilisation

Un élément de tableau reçoit une valeur comme une variable, se lit comme une variable et s'écrit comme une variable. Ce sont dans les structures itératives que les tableaux prennent toutes leurs significations. En effet, les indices des dimensions peuvent être représentés à l'aide de variables.

```
PROGRAMME UTIL
VAR
  notes:tableau[1..10] de réels
  i:entier
DEBUT
  Pour i de 1 à 10 Faire
    Ecrire "Note",i," ?"
    Lire note[i]
  FinPour
  Pour i de 1 à 10 Faire
    Ecrire note[i]
  FinPour
FIN
```

f. Les tableaux dynamiques

Si vous ne connaissez pas par avance le nombre d'éléments de votre tableau, vous avez deux possibilités :

- Fixer un nombre d'éléments suffisamment grand à l'avance pour être sûr d'en avoir assez.
- Ou alors, meilleure solution, redimensionner votre tableau à la bonne taille dès que le nombre d'éléments vous est connu.

Il existe en pseudo-code algorithmique une instruction appelée "Redim" qui permet de redimensionner un tableau dont le nombre d'éléments n'est pas connu à l'avance. Cependant, il est souvent conseillé d'éviter de l'utiliser. Cette instruction trouve son utilité dans le fait qu'en pseudo-code les variables et les tableaux sont déclarés avant le programme, ce qui induit l'impossibilité d'initialiser le nombre d'éléments d'un tableau suivant la valeur d'une variable. Cependant les langages comme Java disposent de mécanismes permettant de déclarer des tableaux sans forcément connaître leur taille à l'avance.

Si vous devez utiliser des tableaux dynamiques, alors vous ne devez pas indiquer de nombres d'éléments dans la déclaration. Cela sera fait dans l'instruction de redimensionnement.

```
PROGRAMME REDIM
VAR
  Elements :tableau[] d'entiers
  Nb:entier
DEBUT
  Ecrire "Combien d'éléments ?"
  Lire nb
  Redim elements[1..nb-1]
FIN
```

Vous ne pouvez pas redimensionner un tableau déjà correctement dimensionné, tout comme il est impossible de dépasser le nombre d'éléments déclarés. Pour obtenir un tableau plus grand, il faut alors en créer un autre, ou utiliser le mécanisme des pointeurs, tel qu'il sera présenté au chapitre Notions avancées.

2. PHP et les tableaux

a. Tableaux classiques à n dimensions

PHP sait gérer des tableaux de 1 à n dimensions. Le principe est presque le même que pour les variables et vous n'êtes donc pas obligé de déclarer un tableau : si un de ses éléments est utilisé, le tableau se met à exister. Les indices ne démarrent pas à 1 mais à 0, il faudra donc adapter le programme en conséquence lors du passage de l'algorithme en PHP. Vous devez utiliser les crochets lors de l'affectation des valeurs et lors de la récupération de celles-ci. Ceux-ci se placent soit après le nom du tableau, soit avant, accolés au type du tableau.

Vous pouvez aussi affecter des valeurs à un tableau avec une fonction (chapitre Les sous-programmes) appelée array.

Une dimension

Vous pouvez déclarer un tableau avec des valeurs prédéfinies comme ceci :

```
$t=array(2,7,9,10,11,14,17,18,20,22);
```

ou encore affecter des index différents :

```
$t=array(1=>2,2=>7,4=>9,6=>10,8=>"toto", "ref"=>"ref01");
```

ou alors donner une valeur à chaque élément indépendamment les uns des autres. Vous accédez au contenu de chaque élément exactement comme prévu, à savoir en mettant le numéro de l'indice entre les crochets.

```
$t[2]=254;
$total=$total+$t[3];
```

Remarquez que ni l'index, ni le contenu d'un tableau ne sont typés. L'index (élément) d'un tableau peut être une chaîne de caractères. C'est ce qu'on appelle un tableau associatif. C'est une fonctionnalité puissante et pratique du PHP très utilisée dans le développement de sites Web dynamiques, notamment pour la récupération de valeurs d'une page à l'autre.

Vous pouvez obtenir le nombre d'éléments d'un tableau à l'aide d'une fonction particulière appelée **count**.

```
$nb=count($t);
```

Références de tableaux

Attention, ici il peut y avoir un risque, notamment si vous venez du langage Java. En Java la copie de tableau se fait par référence. Il est possible en Java de faire ceci :

```
int[] t={2,7,9,10,11,14,17,18,20,22};
int[] copie;
copie=t ;
```

La dernière ligne donne l'impression que le tableau t est copié dans le tableau copie. Or Java fonctionne par référence. Le principe est expliqué dans le point suivant sur la représentation mémoire et dans le chapitre Notions avancées. Ici, ce n'est pas le tableau qui est copié mais copie qui reçoit la référence du tableau t. **Les variables copie et t référencent le même tableau** : si vous modifiez un élément de copie, vous modifiez l'élément correspondant de t puisqu'ils référencent le même tableau, le même endroit dans la mémoire. Ce n'est pas le cas en PHP ou par défaut l'affectation de tableau se fait par copie. L'exemple suivant met ceci en lumière. La première affectation est par copie, la seconde par référence (forcée) : un élément de copie est modifié puis on affiche l'élément correspondant de t : c'est le même.

```
<html>
  <head><meta/>
    <title>tableaux et copies de tableaux</title>
```

```
  </head>
  <body>
  <?php

  $t=array(2,7,9,10,11,14,17,18,20,22);

  // 1 par copie
  $copie=$t;

  echo "$t[2]<br />";
  $copie[2]=5;
  echo "$t[2]<br />";

  // 2 par référence
  $copie=&$t;

  echo "$t[2]<br />";
  $copie[2]=5;
  echo "$t[2]<br />";

?>
  </body>
</html>
```

Tableaux à n dimensions

PHP sait manipuler des tableaux à plusieurs dimensions. Selon le principe précédent, le nom du tableau représente une référence sur le tableau en mémoire. Dans un tableau à n dimensions, chaque dimension référence son propre tableau indépendant en mémoire. Chaque indice de la première dimension référence un tableau pour chaque deuxième dimension. C'est ainsi qu'il est possible que le nombre d'indices de la deuxième dimension (ou de la troisième, quatrième, etc.) ne soit pas le même selon l'indice de la première dimension. La suite vous présente quelques éléments pratiques.

Pour indiquer le nombre d'éléments, procédez comme pour une seule dimension, placez le nombre d'indices entre les crochets.

```
$t1[5][10]=254;
```

Pour initialiser le contenu d'un tableau avec des valeurs prédéfinies, comme pour un tableau à une dimension, utilisez la fonction **array**. Seulement ici vous devez imbriquer plusieurs niveaux, un niveau par dimension, comme ceci :

```
$t1=array(0=>array(10,17,8,9,10,20,13,11,7,5), // 0,0 à 0,9
1=>(9,14,2,0,18,10,16,19,18,6), // 1,0 à 1,9
2=>(17,8,9,7,10,12,11,14,11)); // 2,0 à 2,8
```

La taille de chaque dimension d'un tableau peut être récupérée avec la fonction **count**. Cependant attention, vous n'obtiendrez pas le nombre total d'éléments de tout le tableau, mais pour chaque dimension. L'exemple suivant met ceci en lumière pour un tableau à deux dimensions. Pour obtenir le nombre d'éléments de chaque dimension, récupérez la fonction **count** pour chacune de ces dimensions. La seconde partie de l'exemple fait appel a une structure itérative propre au PHP, **foreach**. Son rôle est de boucler sur chaque élément du tableau, du premier au dernier et d'en isoler tant l'index que la valeur. Ici la valeur est un tableau, puisque la seconde dimension d'un élément donné d'un tableau à deux dimensions est elle-même un tableau d'une dimension.

```
<html>
  <head><meta/>
    <title>2 dimensions</title>
  </head>
  <body>
  <?php
    $total=0;
    $t=array(0 => array ("a","b","c","d"), 1 => array
("e","f","g","h","i"), 2 => array("j","k","l","m","n","o"));
    echo count($t)."<br />";

    for($i=0;$i<count($t);$i++) {
      $total+=count($t[$i]); // calcul nb total d'elements
      echo count($t[$i])."<br />";
    }
```

```
    echo $total;

    echo "<br />";
// seconde possibilité
    $total=0;
    foreach($t as $key) {
      $total+=count($key); // calcul nb total d'elements
      echo count($key)."<br />";
    }
    echo $total;

  ?>
  </body>
</html>
```

3. Représentation en mémoire

a. Représentation linéaire

En principe, les éléments d'un tableau sont placés dans des cases contiguës en mémoire. Si vous prenez un tableau de dix nombres, il pourrait être représenté ainsi :

Case	13121	13122	13123	13124	13125	13126	13127	13128	13129	13130
Indice	1	2	3	4	5	6	7	8	9	10
Valeur	15	17	8	13	10	6	9	13	14	11

La case est le numéro de la case mémoire, l'indice le numéro dans le tableau et la valeur la note associée à l'indice. Une constatation s'impose d'elle-même : les numéros des cases mémoire (les adresses) n'ont pas de rapport avec l'indice, mis à part le fait qu'ils sont contigus.

Autant se représenter un tableau de scalaires à une dimension (un seul indice) en mémoire est simple, autant se représenter deux ou n dimensions devient un peu moins évident, d'autant plus que cette représentation peut varier d'un langage à un autre. Comme le nombre maximal d'indices est connu à l'avance (dans le cas du pseudo-code algorithmique), un tableau à deux dimensions peut être facilement transposé en tableau à une seule dimension.

Soit un tableau note[1..3][1..5] : deux dimensions, qui représentent quinze valeurs. Voici comment ceci pourrait être représenté en mémoire :

Adr	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157
Ind	1,1	1,2	1,3	1,4	1,5	2,1	2,2	2,3	2,4	2,5	3,1	3,2	3,3	3,4	3,5
Val	10	7	14	8	12	11	5	12	13	18	20	2	0	17	16

La première ligne représente l'adresse de la case mémoire associée aux différents indices du tableau. Cette valeur est bien entendu entièrement arbitraire et est connue du langage mettant en œuvre le tableau.

La deuxième ligne représente les indices des différents éléments du tableau. Remarquez que dans cette représentation, on commence par la première dimension, puis par la deuxième, etc. Un tableau à n dimensions peut donc être représenté de manière totalement linéaire. Il est pratique en débutant de se représenter un tableau à deux dimensions en termes de lignes et de colonnes. Mais cette vue de l'esprit est totalement fausse : la mémoire en tant que telle ne peut être représentée ainsi, elle est linéaire (le fameux ruban). Un tableau s'étale donc linéairement dans la mémoire, d'une manière ou d'une autre. Cette dernière remarque est sujette à caution comme vous le verrez un peu plus bas.

Dans le cas du tableau à deux dimensions, comment un langage utilisant ce principe peut connaître la position exacte d'un élément en mémoire ? Soit :

- m la position connue du premier élément,
- x l'indice de la première dimension moins 1,
- y l'indice de la seconde dimension moins 1,

- My la taille maximale de la première dimension.

La position p en mémoire est :

p=m+(x*My)+y

Prenez, d'après le tableau ci-dessus, l'élément d'indice 3,4 : x vaut 2, y vaut 3, Mx vaut 5 et m vaut 143.

p=143+(2*5)+3=156

Soit un nouveau tableau note2[1..2][1..2][1..3], il pourrait être représenté ainsi :

Adr	1512	1513	1514	1515	1516	1517	1518	1519	1520	1521	1522	1523
Ind	1,1,1	1,1,2	1,1,3	1,2,1	1,2,2	1,2,3	2,1,1	2,1,2	2,1,3	2,2,1	2,2,2	2,2,3
Val	10	12	14	10	15	9	8	13	7	5	14	20

Pour calculer la position d'un élément d'indice x,y,z avec les mêmes pré-requis qu'au-dessus, avec Mz la taille maximale de la dimension z et My la taille maximale de la dimension y, on obtient la formule suivante :

p=m+(x*My*Mz)+(y*Mz)+z

b. Représentation par référence

La représentation linéaire ci-dessus est bien pratique pour votre imagination, mais montre ses limites dans certains cas. Notamment, que se passe-t-il avec les types qui ne sont pas des scalaires ? Prenez l'exemple le plus simple : un tableau de chaînes de caractères.

Dans la mémoire et comme vu précédemment, une chaîne de caractères est représentée par une suite de valeurs numériques : les codes ASCII (ou unicode, selon le cas). Le mot "Bonjour" est représenté ainsi :

Lettre	B	o	n	j	o	u	r
ASCII	66	111	110	106	111	117	114

Sachant qu'on ne connaît pas forcément à l'avance la longueur d'une chaîne de caractères, celle-ci se termine souvent, suivant les langages, par un caractère nul. Aussi en mémoire, vous obtiendrez ceci :

Adresse	1616	1617	1618	1619	1620	1621	1622	1623
Contenu	66	111	110	106	111	117	114	0

Cette brève étude met en évidence deux problèmes :

- Dans un tableau à une seule dimension, il n'est pas évident de représenter les indices des chaînes de caractères. Une astuce pourrait consister à rechercher les caractères nuls (0) pour retrouver les indices suivants (le caractère suivant est le premier de la chaîne d'indice +1). Mais :

- La longueur d'une chaîne de caractères n'étant pas fixe, comment réserver à l'avance l'espace contigu nécessaire au stockage de n chaînes dans un tableau de n éléments ?

Vous pourriez évidemment contourner ce problème en décrétant de manière totalement arbitraire que les chaînes de caractères stockées dans votre tableau ont une longueur fixe. Mais quelle perte de place si votre chaîne ne fait que deux caractères pour deux cents réservés ! Ce n'est pas une solution à retenir.

Pour créer des tableaux à n dimensions quelques langages et non des moindres, utilisent une autre méthode. Pour plus de clarté, le mieux est de comprendre le principe tout d'abord avec un tableau à une dimension, puis à deux.

Une chaîne d'une longueur de n caractères est en fait bien souvent un tableau à une dimension comportant n+1 indices, sachant que le dernier indice contiendra un caractère nul. Chaque élément du tableau est le code ASCII (ou unicode) correspondant au caractère de la position (indice) donnée. Ainsi les variables de type Alphanumérique sont, en fonction du langage, des artifices ou plutôt des facilités censées simplifier la vie du développeur. Quand vous affectez une chaîne de caractères à ce type de variable, le langage connaît la longueur de cette chaîne (a="Salut", la longueur de "Salut" est 5) et va allouer l'espace mémoire contigu nécessaire au stockage de cette chaîne. Que représente alors la variable ? Bien souvent ce sera la position en mémoire de la chaîne de caractères,

autrement dit son adresse.

Pour reprendre l'exemple de la chaîne de caractères "Bonjour" ci-dessus, elle débute à l'adresse mémoire 1616. Si c'est la variable "txt" qui "contient" cette chaîne, txt va en fait référencer le tableau de caractères présent à l'adresse 1616. C'est flagrant avec des langages comme le C qui permettent de manipuler directement les adresses mémoire et leur contenu.

Prenez maintenant un tableau de cinq chaînes de caractères :

```
PROGRAMME TABCHAINES
VAR
  Messages:tableau[1..5] de chaînes
DEBUT
  Messages[1]←"il"
  Messages[2]←"ne"
  Messages[3]←"fait"
  Messages[4]←"pas"
  Messages[5]←"beau"
FIN
```

Pour se représenter ceci en mémoire, il faut d'abord se représenter l'organisation des chaînes de caractères. Soit la phrase "il ne fait pas beau", vous voulez placer chacun des mots dans un élément d'un tableau :

Adresses	2007->2009	2010->2012	2013->2017	2018->2021	2022->2026
Contenu	Il	Ne	Fait	Pas	Beau

Notez deux choses :

- Un octet est ajouté pour le caractère nul en fin de chaîne et donc une chaîne de longueur n occupe n+1 octets en mémoire.
- Il se peut que les chaînes ne se suivent pas en mémoire si au moment de les écrire il n'existe pas assez de positions libres contiguës disponibles. Aussi les adresses données dans ce tableau peuvent être totalement décalées par rapport à la réalité.

Le tableau de chaîne de caractères contiendrait donc les références des adresses mémoires où sont réellement stockées les chaînes de caractères :

Indice	0	1	2	3	4
Référence	2007	2010	2013	2018	2022

Un astucieux tour de passe-passe qui permet de nombreuses choses !

Manipulations simples

1. Recherche d'un élément

Vous disposez d'un tableau de n éléments correspondant aux prénoms de vos amis et vous voulez savoir si l'un de ceux-ci est bien présent dans votre tableau. Il faut alors le rechercher. Le principe consiste à balayer l'intégralité du tableau à l'aide d'une structure itérative et à en sortir dès que l'élément a été trouvé ou que le nombre maximal d'indice a été dépassé. À la sortie de la boucle, il faudra de nouveau vérifier pour savoir si oui ou non l'élément a été trouvé : il se peut en effet que tout le tableau ait été parcouru et que ce soit la raison de la sortie de la boucle.

```
PROGRAMME RECHERCHE
VAR
  Tableau noms:tableau[1..10] de chaînes
  rech:chaîne
  i:entier
DEBUT
  i←1
  Tant que i<=10 et noms[i]<>rech Faire
    i←i+1
  FinTantQue
  i←i-1
  Si nom[i]=Rech Alors
    Afficher "Trouvé"
  Sinon
    Afficher "Absent"
  FinSi
FIN
```

Il y a la possibilité de faire différemment avec un drapeau :

```
PROGRAMME RECHERCHE2
VAR
  Tableau noms:tableau[1..10] de chaînes
  Rech:chaîne
  i:entier
  trouve:booléen
DEBUT
  i←1
  trouve←FAUX
  Tant que i<=10 et trouve=FAUX Faire
    Si nom[1]=rech Alors
      trouve←VRAI
    FinSi
    i←i+1
  FinTantQue
  Si trouve Alors
    Affiche "Trouvé"
  Sinon
    Affiche "Absent"
  FinSi
FIN
```

En PHP :

```
<html>
  <head><meta/>
    <title>Recherche</title>
  </head>
  <body>
  <?php

  $t=array(10,20,14,25,17,8,10,12,15,5,41,19,2,6,21);
  $i=0;
  $trouve=false;
  $rech=15;
```

```
  while($i<count($t) && !$trouve) {
    if($t[$i]==$rech) $trouve=true;
    $i++;
  }

  if($trouve) echo "Trouvé à la position ".($i-1)."\n";
  ?>
  </body>
</html>
```

2. Le plus grand/petit, moyenne

Dans le chapitre précédent, vous avez eu l'occasion de déterminer la plus petite et la plus grande d'une série de notes saisies par l'utilisateur. Il s'agit cette fois de faire la même chose avec les tableaux. Le principe est le même sauf que la donnée ne vient pas d'une saisie de l'utilisateur mais du tableau. Voici un exemple pour un tableau de dix éléments :

```
PROGRAMME MINMAXMOY
VAR
  Notes:tableau[1..10] de réels
  min,max,moy:réels
  i:entier
DEBUT
  min←notes[1]
  max←notes[1]
  moy←0
  Pour i de 1 à 10 faire
    Moy=moy+note[i]
    Si note[i]>max Alors
      Max←note[i]
    FinSi
    Si note[i]<min Alors
      Min←note[i]
    FinSi
  FinPour
  Moy←moy/10
  Afficher min,max,moy
Fin
```

En PHP :

```
<html>
  <head><meta/>
    <title>Min, max, moy</title>
  </head>
  <body>
  <?php
  $notes=array(10,20,14,11,17,8,10,12,15,5,16,19,2,6,0);
  $min=$notes[0];
  $max=$notes[0];
  $moy=0;

  for($i=0;$i<count($notes);$i++) {
    $moy+=$notes[$i];
    if($notes[$i]>$max) $max=$notes[$i];
    if($notes[$i]<$min) $min=$notes[$i];
  }
  $moy/=count($notes);

  echo "$min, $max, $moy";
  ?>
  </body>
</html>
```

3. Le morpion

Le jeu du morpion ou tic-tac-toe consiste à aligner des ronds ou des croix en ligne, colonne ou diagonale sur un plateau de 3x3 cases. Le premier joueur qui aligne ses pions a gagné. Sans créer ici une intelligence artificielle pour jouer, l'algorithme va demander à chaque joueur à tour de rôle d'indiquer les coordonnées x (ligne) et y (colonne) où mettre son pion, puis va déterminer si le joueur gagne ou non. Cet algorithme est un peu plus compliqué qu'il n'y paraît :

- Le tableau dispose de deux dimensions, 3x3.
- Il faut au sein d'une boucle permuter les joueurs.
- Si une position est déjà occupée, il faut de nouveau poser la question.
- Après chaque coup, il faut vérifier toutes les lignes, colonnes et diagonales.
- Si une ligne, colonne, diagonale est complète, on sort de la boucle.
- En cas de victoire, il faut indiquer quel pion (x,o) a gagné.
- Il faut gérer le match nul : neuf tours et personne n'a gagné.

Le programme qui suit n'est pas optimisé, de manière tout à fait volontaire, afin de forcer la manipulation des indices de tableaux à deux dimensions :

```
PROGRAMME MORPION
VAR
  p:tableau[1..3][1...3] de caractères
  i,j,x,y,nbtours:entiers
  pion :caractère
  gagne:booléen
DEBUT
  /* Initialisation du plateau : que des blancs */
  Pour i allant de 1 à 3 Faire
    Pour j allant de 1 à 3 Faire
      p[i][j]←" "
    FinPour
  FinPour

  gagne←FAUX
  nbtour←0
  /* Boucle de jeu */
  Répéter

    /* Changement du pion (joueur) */
    Si pion<>"o" Alors
      Pion←"o"
    Sinon
      Pion←"x"
    FinSi

    /* Affichage du plateau */
    Pour i allant de 1 à 3 Faire
      Afficher p[i][1],p[i][2],p[i][3]
    FinPour

/* Saisie des coordonnées */
Répéter
      Afficher "Coordonnées ? (x,y)"
      Saisir x,y
    Jusqu'à x>=1 ET x<=3 ET y>=1 ET y<=3 ET p[x][y]=" "

    /* Mise en place du pion */
    p[x][y] ←pion
```

```
    /* Vérification en ligne */
    i←1
    Tant que i<=3 ET NON gagne
      Si p[i][1]!=" " ET p[i][1]=p[i][2] ET p[i][1]=p[i][3] alors
        gagne←VRAI
      FinSi
      i←i+1
    FinTantQue

    /* Vérification en colonne */
    si NON gagne Alors
    i←1
      Tant que i<=3 ET NON gagne
        Si p[1][i]!=" " ET p[1][i]=p[2][i] ET p[1][i]=p[3][i] alors
          gagne←VRAI
        FinSi
        i←i+1
      FinTantQue
    FinSi

    /* Vérification des deux diagonales */
    Si NON gagne Alors
      Si p[1][1]!=" " ET ((p[1][1]=p[2][2] ET p[1][1]=p[3][3]) OU (p[1][3
]=p[2][2] ET p[1][3]=p[3][1])) Alors
        Gagne←VRAI
      FinSi
    FinSi
    nbtours←nbtours+1 ;
  Jusqu'à gagne=VRAI OU nbtour=9
  Si gagne Alors
    Afficher pion," a gagné !"
  Sinon
    Afficher "Personne ne gagne."
  FinSi
FIN
```

Comme indiqué, ce programme n'est pas optimisé. Ainsi les boucles et tests qui déterminent si les lignes et les colonnes sont gagnantes font appel à des indices statiques. Or si vous souhaitiez par exemple étendre cet algorithme à un "Puissance 4" qui est fondamentalement la même chose, vous auriez des tests à rallonge.

Voici le résultat en PHP, l'affichage ayant été légèrement amélioré et les coordonnées adaptées en fonction des indices des tableaux démarrant à 0.

```
<?php

  $x=0; $y=0; $nbtours=0;
  $pion=' ';
  $tx=""; $ty="";

    /* initialisation du tableau */
  for($i=0;$i<3;$i++) {
    for($j=0;$j<3;$j++) {
      $p[$i][$j]=' ';
    }
  }
  $gagne=false;

  /* boucle principale */
  do {
    if($pion!='o') $pion='o'; else $pion='x';

    /* saisie des coordonnees */
    do {
      /* plateau */
      echo "  1 2 3\n";
      for($i=0;$i<3;$i++) {
        echo ($i+1)."|".$p[$i][0]."|".$p[$i][1]."|".$p[$i][2]."|\n";
      }
      echo "Au tour de $pion\n";
      echo "Coordonnees x";
```

```
      $x=(int)fgets(STDIN)-1;
      echo "Coordonnees y";
      $y=(int)fgets(STDIN)-1;
    }while($x<0 || $x>2 || $y<0 || $y>2 || $p[$x][$y]!=' ');

    $p[$x][$y]=$pion;

    /* ligne */
    $i=0;
    while($i<3 && !$gagne)  {
      if($p[$i][0]!=' ' && $p[$i][0]==$p[$i][1] &&
$p[$i][0]==$p[$i][2]) $gagne=true;
      $i++;
    }

    /* colonne */
    $i=0;
    while($i<3 && !$gagne)  {
      if($p[0][$i]!=' ' && $p[0][$i]==$p[1][$i] &&
$p[0][$i]==$p[2][$i]) $gagne=true;
      $i++;
    }

    /* deux diagonales */
    if($p[1][1]!=' ' &&
      (($p[0][0]==$p[1][1] && $p[0][0]==$p[2][2]) ||
      ($p[0][2]==$p[1][1] && $p[0][2]==$p[2][0]))) $gagne=true;


      $nbtours+=1;

  }while(!$gagne && $nbtours!=9);

  /* plateau */
  echo "  1 2 3\n";
  for($i=0;$i<3;$i++) {
    echo ($i+1)."|".$p[$i][0]."|".$p[$i][1]."|".$p[$i][2]."|\n";
  }

  if($gagne) echo "$pion gagne !";
  else echo "Personne ne gagne.";
?>
```

Algorithmes avancés

1. Les algorithmes des tris

a. Le principe

Vous avez pu voir dans les exemples précédents l'intérêt des tableaux pour le stockage de valeurs multiples. Mais suivant le cas, il peut être utile d'avoir besoin d'obtenir une liste ordonnée de valeurs par ordre croissant ou décroissant. Autrement dit, vous voulez trier le contenu du tableau. Prenez le cas d'un professeur souhaitant trier les notes de ses élèves de la plus basse à la plus haute, ou des résultats d'un tirage du loto pour le rendre plus lisible.

Imaginez un tirage du loto de cinq numéros, évidemment tous différents, dont les valeurs s'étalent entre 1 et 49. Voici l'état initial du tableau suite au tirage au sort :

48	17	25	9	34

Il existe plusieurs méthodes permettant de trier ces différentes valeurs. Elles ont toutes leurs qualités et leurs défauts. Ainsi une méthode sera lente, l'autre sera plus gourmande en mémoire et ainsi de suite. C'est leur complexité qui détermine leur usage notamment pour de grandes plages de valeurs.

Dans les algorithmes suivants, la variable **Cpt** contient le nombre d'éléments du tableau initial et **t[]** est le tableau.

Il est intéressant de prendre en compte la complexité de ces divers algorithmes, bien que cette notion, présentée au premier chapitre, ne soit généralement pas (ou peu) abordée dans les premières années d'études en informatique. Les algorithmes ont souvent une complexité proche. Pourtant à l'usage un tri shell est plus rapide qu'un tri par sélection, tout dépendant du nombre d'éléments et l'éventuel ordre de ceux-ci au départ.

b. Le tri par création

Le tri par création ne sera abordé que du point de vue théorique. En effet si cette méthode semble simple, elle est en fait lourde et compliquée. Si on demande à un débutant en programmation comment trier un tableau, il vous proposera très certainement de créer un deuxième tableau dans lequel on placera au fur et à mesure les éléments du premier tableau dans l'ordre croissant.

C'est une très mauvaise idée pour de multiples raisons dont :

- L'ajout d'un second tableau double la mémoire nécessaire.
- La recherche du plus petit élément est plus compliquée qu'on ne le pense car à chaque passage, il ne faut pas reprendre ceux déjà sortis et c'est compliqué.
- Le nombre de boucles et de recherches est important. La complexité de l'algorithme résultant aussi, supérieure aux autres.

Pour toutes ces raisons, le tri par création ne doit absolument pas être utilisé.

c. Le tri par sélection

Le tri par sélection est très simple : il consiste à sélectionner dans le tableau la plus petite valeur et de la permuter avec le premier élément du tableau, puis la deuxième plus petite valeur (hors premier élément) et de la permuter avec le deuxième élément du tableau et ainsi de suite, et cela pour tous les éléments du tableau. Voici les étapes nécessaires depuis l'exemple général :

48	17	25	9	34

- Étape 1 : la plus petite valeur est 9, on permute 9 et 48.

9	17	25	48	34

- Étape 2 : la plus petite valeur suivante est 17, déjà à la bonne position, on passe à la suivante.

9	17	25	48	34

- Étape 3 : la plus petite valeur suivante est 25, déjà à la bonne position, on passe à la suivante.

9	17	25	48	34

- Étape 4 : la plus petite valeur suivante est 34, on permute 34 et 48. Le tableau est trié.

9	17	25	34	48

Si le principe est simple, l'algorithme résultant nécessite malheureusement la recherche dans tout ou partie du tableau de la plus petite valeur possible et ce sans grande optimisation possible. On peut par contre éviter de permuter des valeurs si aucune valeur inférieure n'a été trouvée. Voici l'algorithme :

```
PROGRAMME SELECTION
VAR
temp,i,j,min,Cpt:entiers
  t:tableau[1..5] d'entiers
DEBUT
  Cpt←5
  Pour i de 1 à Cpt-1 Faire
    min←i
      Pour j de i+1 à Cpt
        Si t[j]<t[min] alors
          min←j
        FinSi
      FinPour
      Si min<>j alors
        temp←t[min]
        t[min] ←t[i]
        t[i] ←temp
      FinSi
    FinPour
FIN
```

À chaque passage dans la boucle, on effectue une comparaison de moins que lors du passage précédent. Le nombre total de passages est donc de *(n-1)+(n-2)+(n-3)* et ainsi de suite soit une complexité de l'algorithme de *n(n-1)/2* ce qui développé donne une complexité d'ordre $O(n^2)$.

Voici le code PHP correspondant :

```
<html>
  <head><meta/>
    <title>tri par selection</title>
  </head>
  <body>
  <?php

  $t=array(27,44,12,18,23,19,101,54,29,77,52,88,10,32);

  $cpt=count($t);
  for($i=0;$i<$cpt-1;$i++) {
    $min=$i;
    for($j=$i+1;$j<$cpt;$j++) {
      if($t[$j]<$t[$min]) $min=$j;
    }
    if($min!=$i) {
      $temp=$t[$min];
      $t[$min]=$t[$i];
      $t[$i]=$temp;
    }
    for($j=0;$j<$cpt;$j++) echo $t[$j]." ";
```

```
    echo "<br />";
  }
?>
  </body>
</html>

}
```

d. Le tri à bulles

Le tri à bulles a un lointain rapport avec le champagne et les boissons gazeuses. Le but est que par permutations successives des valeurs voisines, les valeurs les plus élevées remontent vers les dernières places du tableau, tandis que les valeurs les plus basses migrent vers les premières places. Pour trier dans un ordre croissant, il faut que chaque valeur d'un élément du tableau soit plus petite que celle de l'élément qui suit (sauf pour le dernier, bien entendu). Voici une simulation pas à pas du premier passage :

- Étape 1 : 48 est supérieur à 17, on permute :



17	48	25	9	34

- Étape 2 : 48 est supérieur à 25, on permute :



17	25	48	9	34

- Étape 3 : 48 est supérieur à 9, on permute :

17	25	9	48	34

- Étape 4 : 48 est supérieur à 34, on permute :

17	25	9	34	48

À l'issue de ce premier passage, vous remarquez que la valeur la plus élevée est déjà en dernière place du tableau mais que le tableau n'est pas entièrement trié. Aussi, il faut effectuer plusieurs passages en vérifiant à chaque passage si des permutations ont eu lieu. Quand au moins une permutation a eu lieu lors d'un passage, il faut en relancer un autre. Ainsi, il faut mettre en place un drapeau (flag) indiquant si une permutation a eu lieu ou non. Voici les résultats après les passages successifs :

- Passe 1 :



17	25	9	34	48

- Passe 2 :



17	9	25	34	48

- Passe 3 :



9	17	25	34	48

La structure globale de l'algorithme est donc :

```
PROGRAMME TRIBULLE
VAR
```

```
  Permut :booléen
  temp,Cpt,i:entiers
  t:tableau[1..5] d'entiers
DEBUT
  Cpt←5
  Permut←vrai
  TantQue Permut Faire
    Permut←Faux
    Pour i de 1 à Cpt-1 Faire
      Si t[i]>t[i+1] alors
        temp←t[i]
        t[i]←t[i+1]
        t[i+1]←t[i]
        Permut←Vrai
      FinSi
    FinPour
  FinTantQue
FIN
```

Cependant si vous implémentez cet algorithme dans un quelconque langage, vous allez vous apercevoir que celui-ci dans notre cas précis effectue une passe de trop. En effet, dès le troisième passage, le tableau est trié et pourtant le programme continue. C'est que lors de ce passage, l'algorithme a effectué une permutation des deux premières valeurs 17 et 9. Partant de ce fait, l'indicateur de permutation est passé à Vrai et donc une nouvelle boucle est relancée. Comme il n'est pas possible de prévoir à l'avance le nombre de permutations restantes, l'algorithme montre ses limites dans ce cas précis.

Si n est le nombre d'éléments du tableau, l'algorithme effectue *n-1* boucles *TantQue* et *n-1* boucles *Pour* soit $(n-1)^2$ boucles ce qui se développe en n^2-2n+1. La complexité est d'ordre $O(n^2)$. Autrement dit la complexité de cet algorithme est élevée.

Remarquez aussi que cet algorithme balaie quoi qu'il arrive toutes les valeurs du tableau alors qu'on sait déjà qu'à la première passe la dernière valeur est la plus élevée, qu'à la seconde passe les deux dernières valeurs sont les plus élevées et ainsi de suite. Il est donc possible d'optimiser l'algorithme en décrémentant de 1 la boucle **Pour** à chaque nouvelle passe.

```
...
DEBUT
...
Permut←vrai
Cpt←5
TantQue Permut Faire
  ...
  Pour i de 1 à Cpt-1
    ...
  FinPour
  Cpt←Cpt-1
FinTantQue
FIN
```

La complexité de cet algorithme est un peu moins élevée. En effet on effectue une boucle de moins à chaque passage. La complexité est cependant toujours en $O(n^2)$: au premier passage il y a *(n-1)* comparaisons, au deuxième passage *(n-2)*, au troisième *(n-3)* et ainsi de suite. On obtient donc une complexité de $(n-1)+(n-2)+(n-3)+\ldots+1$ soit $n(n-1)/2$ et donc $n^2-n/2$. C'est identique au tri par sélection.

Le code PHP correspondant est le suivant :

```
<html>
  <head><meta/>
    <title>tri bulle</title>
  </head>
  <body>
  <?php

  $t=array(14,13,12,11,10,9,8,7,6,5,4,3,2,1);

  $Permut=true;

  $cpt=count($t)-1;

  while($Permut) {
    for($i=0;$i<count($t);$i++) echo $t[$i]." ";
```

```
    echo "<br />";
    echo "->";
    Permut=false;

    for($i=0;$i<$cpt;$i++) {
      if($t[$i]>$t[$i+1]) {
        $temp=$t[$i];
        $t[$i]=$t[$i+1];
        $t[$i+1]=$temp;
        $Permut=true;
      }
    }
    $cpt--;
    for($i=0;$i<count($t);$i++) echo $t[$i]." ";
    echo "<br />";
  }
?>
  </body>
</html>
```

e. Le tri par insertion

Le tri par insertion consiste à sélectionner un élément du tableau et à l'insérer directement à la bonne position dans la partie du tableau déjà triée. On procède en trois étapes :

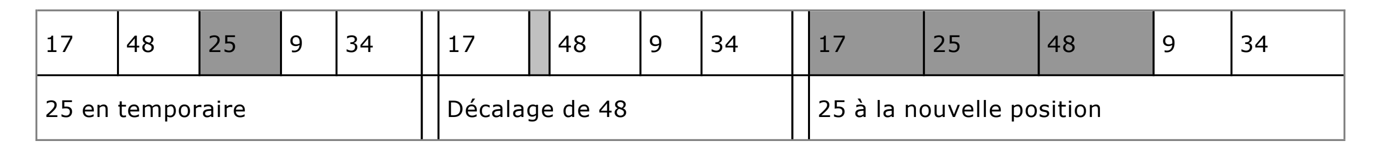
- On place l'élément à trier dans une variable temporaire.
- Tant que les éléments du tableau qui précèdent l'élément à trier lui sont supérieurs, on décale ces éléments d'une position en récupérant l'espace vide laissé par l'élément à trier.
- On insère ensuite la variable temporaire à la nouvelle position laissée vacante par le décalage.

Voici les différentes étapes pour le tableau exemple :

- **Étape 1** : le deuxième élément 17 est placé dans une variable temporaire qui est comparée aux éléments qui le précèdent. Chacun est décalé tant qu'il est supérieur à l'élément à trier.

48	17	25	9	34		48	25	9	34	17	48	25	9	34
17 en temporaire					Décalage de 48					17 à la nouvelle position				

- **Étape 2** : 25 est comparé aux éléments qui le précédent et chacun est décalé jusqu'à ce que l'élément ne soit plus supérieur au troisième.



17	48	25	9	34	17		48	9	34	17	25	48	9	34
25 en temporaire					Décalage de 48					25 à la nouvelle position				

- **Étape 3** : 9 est comparé aux éléments qui le précèdent. Ici comme dans l'étape 1 on s'arrête forcément au premier élément.

17	25	48	9	34		17	25	48	34	9	17	25	48	34
9 en temporaire					Décalage de 17, 25 et 48					9 à la nouvelle position				

- **Étape 4** : 34 est comparé aux éléments qui le précèdent. Seul 48 lui est supérieur.

9	17	25	48	34	9	17	25		48	9	17	25	34	48
34 en temporaire					Décalage de 48					34 à la nouvelle position				

L’algorithme résultant est assez simple. Seule la boucle de décalage peut être un peu plus ardue à comprendre. Chaque élément est décalé vers la droite (ou le bas selon la représentation qu’on s’en fait) du tableau tant qu’il est supérieur à l’élément recherché.

```
PROGRAMME TRINSERTION
VAR
  i,mem,pos:entiers
  t:tableau[1..5] d’entiers
DEBUT
  Cpt←5
  Pour i de 1 à Cpt faire
    mem←t[i]
    pos←i-1
    tant que pos>=0 ET t[pos]>mem Faire
      t[pos+1]←t[pos]
      pos←pos-1
    FinTantQue
    t[pos+1]←mem
  FinPour
FIN
```

Comme souvent, la complexité varie selon l’ordre initial des éléments dans le tableau à trier. Cependant dans le pire des cas, on effectue *(n-1)* boucles dans lesquelles on effectue une moyenne de *(n-2)/2* échanges et donc un total de (*n-1)(n-2)/2*. On obtient une complexité d’ordre *O(n²)*. Cependant on effectue en moyenne seulement la moitié des comparaisons (dans l’exemple ci-dessus, six comparaisons sont effectuées alors que dix auraient pu être effectuées). La complexité est alors bien moindre. Dans la pratique, un tri par insertion est généralement plus rapide que les tris à bulles et par sélection.

Une petite remarque concernant le code PHP. Si vous faites :

```
while($t[$pos]>$mem && $pos>=0)
```

L’expression est évaluée de gauche à droite. Vous allez avoir une erreur à un moment donné : quand pos vaut 0, à la boucle suivante il vaut -1. Or si un langage comme le C permet de déborder les indices (aucune vérification n’est effectuée), Java et PHP ne le permettent pas et causent une exception qui stoppe le programme avec une erreur. Aussi, il faut d’abord tester la valeur de pos AVANT de vérifier le contenu du tableau à cet indice.

```
while($pos>=0 && $t[$pos]>$mem)
```

Le code correspondant en PHP est le suivant :

```
<html>
  <head><meta/>
    <title>tri par insertion</title>
  </head>
  <body>
  <?php

  $t=array(48,17,25,9,34);
  $cpt=5;

  for($i=1;$i<$cpt;$i++) {
    $mem=$t[$i];
    $pos=$i-1;
    while(($pos>=0) && ($t[$pos]>$mem)) {
      $t[$pos+1]=$t[$pos];
      $pos--;
    }
    $t[$pos+1]=$mem;
    for($j=0;$j<$cpt;$j++) echo $t[$j]." ";
    echo "<br />";
  }
?>
  </body>
```

```
</html>
```

f. Le tri Shell

Le tri Shell est une variante du tri précédent qui a été proposé par Donald L. Shell en 1959 (il n'y a donc aucun rapport avec le shell Unix ou Windows). Dans ce type de tri, les éléments ne sont plus décalés de un à un mais par pas plus important. La permutation s'effectue en fonction de ce pas. Une fois les permutations de ce pas effectuées, le pas est réduit. Quand le pas atteint 1, le tri Shell devient un "bête" tri par insertion. Au final, le tri Shell consiste à dégrossir un maximum le tableau à trier en plaçant dès les premiers passages le plus d'éléments possibles dans les bonnes parties du tableau. Dans un tableau d'une dizaine d'éléments, la moyenne des éléments de la première partie du tableau est plus basse que celle de la deuxième partie, dès le premier passage.

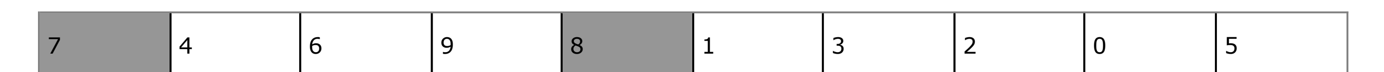
Au final, le tri Shell est d'une complexité $O(n^2)$ mais se révèle être plus rapide dans la majorité des cas. C'est l'algorithme de tri le plus utilisé.

Prenez un tableau de dix éléments :

8	4	6	9	7	1	3	2	0	5

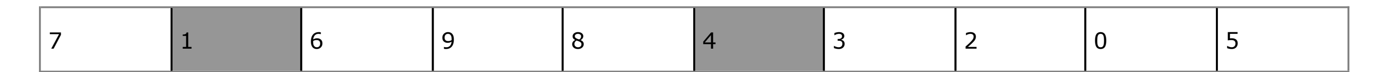
Et un pas de 4 :

- **Étape 1** : t[1] et t[5] sont comparés et éventuellement permutés :



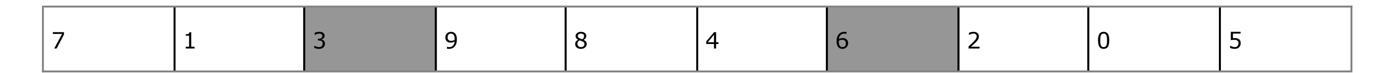
7	4	6	9	8	1	3	2	0	5

- **Étape 2** : t[2] et t[6] sont comparés et éventuellement permutés :



7	1	6	9	8	4	3	2	0	5

- **Étape 3** : t[3] et t[7] sont comparés et éventuellement permutés :



7	1	3	9	8	4	6	2	0	5

- **Étape 4** : t[4] et t[8] sont comparés et éventuellement permutés :

7	1	3	2	8	4	6	9	0	5

- **Étape 5** : t[5] et t[9] sont comparés et éventuellement permutés :

7	1	3	2	0	4	6	9	8	5

...

Le pas ne doit pas être calculé au hasard car c'est de lui que dépend l'efficacité de l'algorithme. La formule utilisée par l'algorithme est généralement :

$U_{(n+1)}=3_{Un}+1$ avec $U_0=0$

```
PROGRAMME TRISHELL
VAR
  cpt,n,i,j,tmp:entiers
  t:tableau[1..10] d'entiers
DEBUT
  cpt←10
  n←0
  TantQue n<cpt Faire
    n←3*n+1
  FinTantQue
```

```
    TanQue n<>0 Faire
      n←n/3
      Pour i de n à cpt-1 Faire
        tmp←t[i]
        j←i
        TantQue j>n-1 ET t[j-n]>tmp
          t[j] ←t[j-n]
          j←j-n
        FinTantQue
        t[j] ←tmp
      FinPour
    FinTantQue
  FIN
```

Soit en PHP :

```
<html>
  <head><meta/>
    <title>tri shell</title>
  </head>
  <body>
  <?php

  $t=array(48,17,25,9,34,12,28,1,4,98,0,33,48,10,11,9,25);

  $n=0;
  $cpt=count($t);

  while($n<$cpt) $n=3*$n+1;

  while($n!=0) {
    $n=(int)($n/3);
    for($i=$n;$i<$cpt;$i++) {
      $mem=$t[$i];
      $j=$i;
      while($j>($n-1) && $t[$j-$n]>$mem) {
        $t[$j]=$t[$j-$n];
        $j=$j-$n;
      }
      $t[$j]=$mem;
    }
    for($j=0;$j<$cpt;$j++) echo $t[$j]." ";
    echo "<br />";
  }
?>
  </body>
</html>
```

2. Recherche par dichotomie

La recherche par dichotomie ne s'applique que sur les tableaux déjà triés. Vous avez déjà rencontré un algorithme de recherche d'élément dans un tableau non trié. Mais celui-ci pose un problème : si le tableau contient 10 000 éléments et que par pur hasard celui que vous cherchez est le 10000ème, il faut balayer l'intégralité du tableau. Cette recherche séquentielle n'est pas idéale.

Dans un tableau trié, la problématique est radicalement différente. Rien qu'avec une recherche séquentielle, il devient inutile de balayer tout le tableau : il suffit de s'arrêter dès que la valeur de l'élément du tableau devient supérieure à ce qu'on recherche, d'où une probable complexité moyenne plus basse. Mais il reste une solution plus efficace.

La dichotomie consiste à diviser par deux l'intervalle de recherche tant que l'élément recherché n'est pas trouvé. Sur un tableau t de 10 éléments triés :

Indice	1	2	3	4	5	6	7	8	9	10
Valeur	2	7	9	10	11	14	17	18	20	22

Vous voulez savoir si la valeur 20 est présente dans le tableau.

- **Étape 1** : Calculer l'indice situé au milieu du tableau. L'indice de début est 1, l'indice de fin est 10, le milieu vaut début+fin/2. Comme cette valeur n'est pas forcément entière on récupère la division entière : (début+fin) DIV 2. Ici 5.

Indice	1	2	3	4	5	6	7	8	9	10
Valeur	2	7	9	10	11	14	17	18	20	22

- **Étape 2** : Comparer la valeur t[5] avec 20. Étant inférieure, cela veut dire que la valeur 20 est forcément au-delà de l'indice 5. On positionne Début à 6 et on recalcule (début+fin) DIV 2. Ici 8.

Indice	1	2	3	4	5	6	7	8	9	10
Valeur	2	7	9	10	11	14	17	18	20	22

- **Étape 3** : Comparer la valeur t[8] avec 20. Étant inférieure, cela veut dire que la valeur 20 est au-delà de l'indice 8. On positionne Début à 9 et on recalcule. On obtient 9.

Indice	1	2	3	4	5	6	7	8	9	10
Valeur	2	7	9	10	11	14	17	18	20	22

- **Étape 4** : Comparer t[9] avec 20. Les valeurs sont identiques, la recherche est terminée.

La recherche doit continuer tant que Début est inférieur ou égal à Fin et que l'élément recherché n'a pas été trouvé.

```
PROGRAMME DICHOTOMIE
VAR
  t:tableau[1..10] d'entiers
  d,m,f,rech:entiers
DEBUT
  rech←18
  d←1
  f←10

  Répéter
    m←(d+f) DIV 2
    Si rech>t[m] Alors
      d←m+1
    Sinon
      f←m-1
    FinSi
  TantQue d<=f ET rech<>t[m]
  Si rech=t[m] Alors
    Afficher "Trouvé"
  Sinon
    Afficher "Absent"
  FinSi
FIN
```

Le code correspondant en PHP est le suivant :

```
<html>
  <head><meta/>
    <title>Recherche dichotomique</title>
  </head>
  <body>
  <?php

  $t=array(2,7,9,10,11,14,17,18,20,22);
```

```
  $rech=14;
  $d=0;
  $f=count($t)-1;

  do {
    $m=(int)(($d+$f)/2);
    echo "d=$d, f=$f, m=$m t[m]={$t[$m]}<br />";
    if($rech>$t[$m]) $d=$m+1;
    else $f=$m-1;

  } while($d<=$f && $rech!=$t[$m]);

  if($rech==$t[$m]) echo "$rech trouvé à la position $m";
  else echo "$rech n'a pas été trouvé";
  ?>
  </body>
</html>
```

Structures et enregistrements

1. Principe

Les tableaux sont certes très pratiques, mais ils ne permettent pas toujours de répondre efficacement à tous les besoins de stockage. Un tableau est une structure de données dont tous les éléments sont de même type. Que faire quand vous avez besoin de placer dans une structure de type tableau des enregistrements de types différents ?

Comme exemple concret, prenez un catalogue de produits dans un magasin spécialisé. Un article est décrit à l'aide d'une référence, un nom (libellé) et un prix. Les deux premiers sont des chaînes de caractères, le dernier un nombre réel. Comment se représenter cela avec des tableaux ? Il faudrait trois tableaux : un pour les références, un autre pour les libellés et un troisième pour les prix. L'indice de l'article devrait être identique pour les trois tableaux.

C'est possible, faisable, mais en pratique totalement ingérable dès qu'il s'agit d'aller un peu plus loin que de simples traitements. Quid des tri ? Quid des recherches ? Cela devient difficile. Il faudrait donc une sorte de méta-type particulier qui pourrait regrouper en un seul ensemble des variables de types différents.

Ces méta-types existent. Ils s'appellent des structures, ou types structurés et permettent de décrire des enregistrements. Les enregistrements sont en fait des structures de données composées d'éléments de types différents ou non. Ces structures composées de plusieurs éléments forment une entité unique qui est appelée un **type structuré**.

Autrement dit, vous pouvez créer vos propres types de données en combinant d'autres éléments de types différents ou non et créer des variables de ce nouveau type, qu'on appelle des enregistrements. Les différents éléments contenus dans un type structuré sont appelés des **champs**.

2. Déclaration

a. Type structuré

Le type structuré est opposable aux types dit primitifs vus jusqu'à présent. Un type structuré peut contenir des éléments de types primitifs (entiers, réels, chaînes, caractères), des tableaux, mais aussi des éléments d'autres types structurés. Ceci permet une infinité de nouveaux types, pour tous les cas de figures.

Un type structuré doit être déclaré et défini avant les variables pour qu'il puisse être utilisé pour définir des enregistrements. Le type structuré se déclare donc entre les constantes et les variables. Si l'algorithme contient des sous-programmes, vous déclarerez les types structurés hors du programme et des sous-programmes, c'est-à-dire tout en haut de celui-ci. La structure est déclarée dans une section particulière sous le mot-clé "**Type**", entre les mots-clés **Structure** et **FinStruct**.

```
Type
  Structure nom_type
    champ1:type_champ1
    champ2:type_champ2
    ...
    Champn :type_champn
  FinStruct
```

- Chaque structure porte un nom. Ce nom sera utilisé pour déclarer des enregistrements.
- Une structure peut contenir 1 à n champs, du même type ou de types différents. Une structure à un seul champ est en soi totalement inutile.

La structure pour décrire un article serait donc quelque chose comme :

```
Type
  Structure tarticle
    ref:chaîne
    libelle:chaine
    prix:réel
  FinStruct
```

b. Enregistrement

Un enregistrement est une "variable" d'un type structuré donné. Il se déclare exactement comme une variable, au même endroit, sous le mot-clé VAR. Un enregistrement peut donc être considéré comme une variable, un peu spéciale cependant.

```
VAR
  nom_enreg:nom_type
```

Dans le cadre de l'exemple précédent, vous déclarez des articles ainsi :

```
VAR
  article1,article2,article3:tarticle
```

En mémoire, les divers éléments d'un enregistrement peuvent généralement être représentés comme étant dans des zones contiguës.

article1.ref article1.libelle article1.prix
article1
article1.ref article1.libelle article1.prix
article2
article1.ref article1.libelle article1.prix
article3

Avec un enregistrement de ce type, il existe des ressemblances évidentes avec l'éventuelle structure d'enregistrements en base de données ou dans un fichier. Pour peu que le type structuré de l'enregistrement reprenne le même schéma, les traitements s'en trouvent grandement simplifiés.

Cependant, particulièrement pour la ressemblance avec un contenu de base de données relationnelle, l'analogie n'est pas complète. Un enregistrement ne contient pas d'identifiant ou clé uniques et rien n'empêche que deux enregistrements de même type contiennent les mêmes données. Ce serait à vous au sein de votre programme de gérer ces cas de figure.

De même, le champ d'application des enregistrements est beaucoup plus vaste qu'il n'y paraît. Certains langages les utilisent pour des choses totalement différentes qu'une description de données de gestion. Ainsi un langage comme le C dispose de types structurés pour gérer les fichiers : ouverture, fermeture, position, type d'accès, etc., mais aussi leur nom, leur référence sur le disque, leurs propriétaires, leurs droits… D'autres types structurés sont utiles pour la gestion des dates et heures, pour représenter une connexion réseau…

3. Utiliser les enregistrements

Les enregistrements sont composés de plusieurs éléments appelés champs. Quand vous manipulez un enregistrement, vous le faites au travers de ses champs :

- Il n'est pas possible d'affecter une valeur à un enregistrement en passant par son nom. Pour lui affecter des valeurs, il faut les affecter une à une aux champs correspondants.
- Cependant, il est possible d'affecter un enregistrement à un autre du même type : chaque champ de l'enregistrement affecté reçoit la valeur du champ correspondant de l'enregistrement à affecter.

a. Utiliser les champs

Vous accédez aux champs d'un enregistrement en passant par le nom de l'enregistrement et le nom du champ séparés par le caractère ".", le point, selon la forme suivante :

```
nom_enreg.nom_champ
```

Cette syntaxe représente la valeur du champ *nom_champ* au sein de l'enregistrement *nom_enreg*. Reprenons l'exemple précédent :

```
article1.prix
```

représente le prix de l'article1. D'ailleurs, pour plus de simplicité, lisez vos enregistrements de droite à gauche : article2.libellé se lit "libellé de article2".

Comme il est interdit d'affecter une valeur directement à un enregistrement complet, vous devez passer par les champs. Évidemment il est impossible, si vous mettez le nom du champ seul, de savoir à quel enregistrement il appartient. Aussi n'oubliez jamais d'écrire le nom de l'enregistrement ET le nom du champ. De même, ne confondez pas le nom du type structuré et le nom de l'enregistrement :

```
tarticle.ref /* mauvaise idée */
```

ne représente rien du tout et ne peut recevoir aucune valeur.

Les champs d'un enregistrement se manipulent exactement comme des variables, ils peuvent recevoir des valeurs et leur valeur peut être affectée à une autre variable. Les champs peuvent être utilisés partout où les variables sont utilisées, y compris, vous le verrez dans le prochain chapitre, comme paramètres de sous-programmes, en saisie, en affichage, etc.

L'exemple suivant reprend tous ces principes :

```
PROGRAMME demo_enreg
Type
  Structure tarticle
    ref:chaîne
    libelle:chaine
    prix:réel
  FinStruct
Var
  article1,article2:tarticle
  reponse:chaîne
DEBUT
  Afficher "Référence du premier article ?"
  Saisir article1.ref
  Afficher "Libellé du premier article ?"
  Saisir article1.libellé
  Afficher "Prix du premier article ?"
  Saisir article1.prix
  Afficher article1.ref,article1.libelle,article1.prix
  Afficher "Copier le premier article dans le second ?"
  Saisir reponse
  Si reponse="oui" Alors
    article2=article1
    Afficher article2.ref,article2.libelle,article2.prix
  FinSi
  article2.prix←15.25
  Si article1.prix=article2.prix Alors
    afficher "Les deux articles ont le même prix"
  FinSi
FIN
```

Voici les constatations qui peuvent être tirées de cet exemple :

- Le nom du champ seul ne représente rien, il est toujours associé à son enregistrement sous la forme *enreg.champ*. Ce qui est appelé champ représente cette paire.
- Un champ s'utilise exactement comme une variable indépendante. C'est ce qu'elle est puisqu'un type structuré est un ensemble de variables d'un type donné.
- De ce fait, un champ peut recevoir une valeur comme une autre.
- Enfin, le seul cas où un enregistrement peut recevoir une valeur en globalité c'est quand on lui affecte un autre enregistrement du même type.

b. Un enregistrement dans une structure

Un type structuré définit un nouveau type de variable appelé enregistrement. Un enregistrement est déclaré comme une variable. Il semble donc logique qu'un enregistrement puisse faire lui-même partie d'un autre type structuré.

Chaque article dispose d'un fabricant. Ce fabricant peut être décrit par une structure typée :

```
Structure tfabricant
  ref:chaîne
  nom:chaîne
  adresse:chaîne
  tel:chaîne
FinStruct
```

Vous voulez maintenant associer un fabricant à chacun de vos articles. Rien de plus simple : incorporez un enregistrement de type tfabricant à votre type structuré tarticle :

```
Structure tarticle
  ref:chaîne
  libelle:chaîne
  prix:chaîne
  fab:tfabricant
FinStruct
```

Maintenant déclarez un enregistrement de type article :

```
VAR
  art:article
```

Dans votre programme, vous allez accéder aux champs de l'enregistrement art comme vu ci-dessus, mais vous allez aussi ajouter les informations sur le fournisseur. Pour cela, il suffit de respecter la syntaxe avec des points :

```
nom_enreg1.nom_enreg2.nom_champ
```

Soit comme exemple :

```
Début
  art.ref←"art001_01"
  art.libelle←"Pelle à tarte Inox Luxe"
  art.prix←35.50 /* cher pour une pelle à tarte */
  art.fab.ref←"Fab1234"
  art.fab.nom←"Le roi de la tarte"
  art.fab.adresse←"12 rue Siflette 13248 Latruelle"
  art.fab.tel←"0404040404"
Fin
```

c. Un tableau dans une structure

Vous voulez maintenant connaître le nombre d'articles vendus sur les douze mois de l'année. Pour cela, vous allez créer un nouveau type structuré qui va reprendre un enregistrement tarticle auquel vous allez ajouter un moyen de stocker douze valeurs réelles. Le meilleur moyen que vous connaissez déjà est le tableau. Vous pouvez parfaitement ajouter un tableau comme champ de structure.

```
Structure bilanart
  art:tarticle
  vente:tableau[1..12] de réels
FinStruct
```

Soit dit en passant, dans une telle structure et un enregistrement bart1 associé, l'accès aux divers champs peut devenir un peu long. Vous accédez au tableau vente ainsi :

```
bart1.vente[indice]
```

L'algorithme suivant va demander la saisie successive des douze quantités de ventes mensuelles :

```
TYPES
  // Le fabricant
Structure tfabricant
    ref:chaîne
  nom:chaîne
    adresse:chaîne
```



```
  tel:chaîne
FinStruct
  // L'article
Structure tarticle
    ref:chaîne
    libelle:chaîne
    prix:chaîne
    fab:tfabricant
FinStruct
  // Les ventes mensuelles
Structure bilanart
    art:tarticle
    vente:tableau[1..12] de réels
FinStruct
VAR
  bart01:bilanart
  i,total=0:entiers
Début
  bart01.art.ref←"art001_01"
  bart01.art.fab.ref←"Fab1234"
  Pour i de 1 à 12 Faire
    Afficher "Ventes du mois",i," ?"
    Saisir bart01.vente[i]
    total←total+bart01.vente[i]
  FinPour
  Afficher "Total annuel :" ,total
Fin
```

4. Les tableaux d'enregistrements

a. Les tables

Un article représente un enregistrement. Jusqu'à présent pour représenter plusieurs articles, vous deviez créer plusieurs enregistrements, comme dans l'exemple de article1, article2 et article3. L'idéal serait de pouvoir traiter n articles sans avoir à déclarer n enregistrements indépendants. Un enregistrement étant (la répétition est le meilleur ami de la mémoire) déclaré comme une variable, vous avez aussi le droit de créer des tableaux d'enregistrements. Un tableau d'enregistrements se déclare comme n'importe quel autre tableau. Il est parfois appelé **table**. Dans cette table, les colonnes sont les champs et les lignes les enregistrements.

Soit la structure tarticle simplifiée d'origine :

```
Structure tarticle
  ref:chaîne
  libelle:chaine
  prix:réel
FinStruct
```

Vous voulez créer une table de dix enregistrements :

```
Var
  articles :tableau[1..10] de tarticles
```

Pour saisir les éléments d'un enregistrement de cette table, vous utilisez les indices :

```
articles[1].ref="art001_01"
```

Pour accéder aux dix enregistrements, le mieux est d'utiliser une boucle :

```
Début
  Pour i de 1 à 10 Faire
    Afficher "Saisir ref article",i
    Saisir articles[i].ref
  FinPour
Fin
```

b. Une table comme champ

Selon le même principe qu'un tableau dans une structure, vous pouvez utiliser un tableau d'enregistrements comme type de données dans une structure, puisque ce n'est qu'un tableau, après tout.

Soit une grande enseigne disposant de dix magasins. Vous devez décrire deux structures : une structure magasin et une structure enseigne. La structure enseigne doit contenir les enregistrements des dix magasins. Voici ce que vous pourriez faire :

```
Structure tmagasin
  adresse:chaîne
  tel:chaîne
  gerant:chaîne
FinStruct
Structure tenseigne
  nom:chaîne
  magasin :tableau[1..10] de tmagasins
FinStruct
```

Voici comment exploiter un enregistrement de type tenseigne :

```
Var
  enseigne :tenseigne
  i :entier
Début
  Afficher "Nom de l'enseigne ?"
  Saisir enseigne.nom
  pour i de 1 à 10 Faire
    Afficher "adresse du magasin",i
    Saisir enseigne.magasin[i].adresse
    Afficher "tel du magasin",i
    Saisir enseigne.magasin[i].tel
    Afficher "gérant du magasin",i
    Saisir enseigne.magasin[i].gerant
  FinPour
Fin
```

Il n'y a en fait rien de bien complexe. Vous pouvez encore aller plus loin en créant des tables contenant d'autres tables et ainsi de suite...

5. Et PHP ?

Il y a ici un petit problème. PHP ne dispose pas, tout au moins directement, de moyen de déclarer une structure en tant que telle, contrairement à des langages comme le C, C++ ou le Pascal. PHP est un langage dit objet. L'objet est abordé dans le dernier chapitre. Pour créer un objet PHP, il faut écrire une classe : c'est la description ou une sorte de "type" d'objet. Une classe peut contenir des variables (les attributs) et des bouts de programme (les méthodes). Par (mauvaise) analogie une classe qui ne contiendrait que des variables pourrait être apparentée à une structure. Voici un exemple succinct :

```
<html>
  <head><meta/>
    <title>Structures</title>
  </head>
  <body>
  <?php

class tfabricant {
  public $ref;
  public $nom;
  public $adresse;
  public $tel;
}
class tarticle {
  public $ref;
  public $libelle;
  public $prix;
  public $fab;
```

```
}

  $article[0]=new tarticle;
  $article[1]=new tarticle;

  $article[0]->fab=new tfabricant;
  $article[1]->fab=new tfabricant;

  $article[0]->ref="Art001_01";
  $article[1]->ref="Art002_02";

  $article[0]->fab->ref="Fab1234";
  $article[1]->fab->ref="Fab4321";

  echo $article[0]->ref;
  echo "<br />";
  echo $article[0]->fab->ref;
  echo "<br />";
  echo $article[1]->ref;
  echo "<br />";
  echo $article[1]->fab->ref;
?>
  </body>
</html>
```

Il est fortement probable que vous ne compreniez pas certaines instructions comme public, new, pourquoi class et pas struct, etc. Ne vous inquiétez pas, tout ceci sera expliqué au dernier chapitre consacré à l'objet. Pour l'instant, remarquez seulement que dans ce cas précis, le mot-clé **class** peut être utilisé pour déclarer des sortes de structures. Il est aussi possible de créer des tableaux d'objets :

```
<html>
  <head><meta/>
    <title>Structures 2</title>
  </head>
  <body>
  <?php

class tfabricant {
  public $ref;
  public $nom;
  public $adresse;
  public $tel;
}
class tarticle {
  public $ref;
  public $libelle;
  public $prix;
  public $fab;
}

  for($i=0;$i<3;$i++) {
    $article[$i]=new tarticle;
    $article[$i]->fab=new tfabricant;
  }

  $article[0]->ref="Art001_01";
  $article[0]->fab->ref="Fab1234";
  $article[1]->ref="Art002_02";
  $article[1]->fab->ref="Fab4321";

  echo $article[0]->ref;
  echo "<br />";
  echo $article[0]->fab->ref;
  echo "<br />";
  echo $article[1]->ref;
  echo "<br />";
  echo $article[1]->fab->ref;
?>
  </body>
```

```
</html>
```

Présentation

1. Principe

Lors de la présentation de la structure d'un algorithme, il a été brièvement abordé la possibilité d'ajouter une partie supplémentaire en début de programme, avant les déclarations des variables. Cette partie n'avait pas encore été abordée, alors qu'à ce niveau vous savez, si vous avez bien compris les chapitres précédents, déjà très bien programmer. Or peut-être avez-vous remarqué quelques limitations frustrantes et notamment une certaine lourdeur lorsque le programme est très long et qu'il s'agit de répéter certains blocs d'instructions pourtant déjà présents ailleurs. Par exemple, rappelez-vous un simple petit programme qui calcule la valeur absolue d'une commande. L'embêtant est qu'à chaque fois que vous voulez calculer cette valeur, vous devez répéter la structure conditionnelle. N'aurait-il pas été plus simple de le faire une seule fois et de passer à ce bloc d'instructions uniquement la valeur dont on veut récupérer la valeur absolue ?

Vous pourriez pour cela envisager un second programme qui serait lancé par le programme principal avec comme paramètre cette valeur. C'est techniquement faisable, mais placer un second programme à part (un exécutable) juste pour ce genre de traitement, c'est une perte de temps et d'espace. L'autre solution consiste à ajouter le code nécessaire à ce programme dans une structure spéciale et à part du programme principal. C'est ce qu'on appelle un sous-programme. Les anciens langages BASIC utilisaient d'ailleurs des instructions en ce sens (gosub, sub xxx, endsub, etc, sub pour sous-programme).

Lorsqu'un programme est très long, il n'est pas réaliste de tout programmer d'un seul tenant. Le programme est décomposé en de plus petites unités ou parties réutilisables qui sont ensuite appelées au moment opportun par le programme principal. Un sous-programme évite la répétition inutile de code et permet de clarifier le programme. Une fois tous les sous-programmes réalisés, il est même possible de les enregistrer dans des bibliothèques pour les réutiliser dans d'autres programmes, quels qu'ils soient, simplifiant ainsi fortement l'écriture du code et permettant aussi d'aller beaucoup plus vite.

Un programme complet forme une application. Une application est, si vous avez correctement suivi, composée de plusieurs parties fonctionnelles :

- Le programme principal, ou bloc principal, qui correspond au bloc principal d'instructions situées sous le mot-clé PROGRAMME et DEBUT. C'est ce programme qui est exécuté quand vous lancez l'exécutable qui résulte de l'implémentation de votre algorithme en Java, par exemple. En Java, le programme principal est ce qui est situé sous la ligne contenant le mot "main" qui signifie depuis l'anglais "principal".

- Des sous-programmes, chargés de divers rôles, qui peuvent aller du calcul d'une valeur absolue à celui d'une puissance quelconque, une conversion de date, le formatage d'une chaîne de caractères, l'affichage d'un en-tête quelconque, bref tout ce que vous voudrez bien en faire. Suivant les langages, vous trouverez les expressions sous-programmes, mais aussi et surtout les mots **fonctions** et **procédures** pour les décrire. C'est le programme principal qui se charge d'appeler les sous-programmes. Ceux-ci ne se lancent jamais d'eux-mêmes.

Un sous-programme ne se lance jamais tout seul, il doit être appelé depuis le programme principal. Cependant, un sous-programme peut parfaitement faire appel à un autre sous-programme. Par exemple, un sous-programme chargé de calculer une racine carrée peut appeler le sous-programme chargé des valeurs absolues…

---

2. Déclaration et définition

a. Dans un algorithme

Avant de pouvoir utiliser un sous-programme, il faut le définir ou le déclarer, c'est-à-dire indiquer au programme principal qu'il existe : son nom et son contenu (bloc d'instructions). En algorithmique, les sous-programmes sont déclarés et entièrement écrits au tout début, avant le mot-clé PROGRAMME. C'est généralement le cas dans les langages de programmation, car le programme principal ne peut pas utiliser un sous-programme s'il ne sait pas qu'il existe.

```
<SOUS-PROGRAMME 1>
<SOUS-PROGRAMME 1>
...
PROGRAMME XYZ
```

```
VAR
...
DEBUT
...
FIN
```

Si dans votre algorithme vos sous-programmes sont écrits en dessous du programme principal, il est fort probable quand vous passerez à la programmation, que vous obteniez des erreurs. Un langage comme le C autorise cependant la déclaration du nom du sous-programme en haut du programme et sa programmation en dessous. Pour le PHP, peu importe l'endroit où le sous-programme est écrit. Quant à Java, la problématique est différente du fait de sa conception objet (dernier chapitre).

Le sous-programme a une structure assez simple, c'est la même que pour le programme principal : vous y déclarez vos variables, constantes, tableaux, etc. et vous placez vos instructions entre DEBUT et FIN. Voici un simple exemple, pour le moment ayez une confiance aveugle, les termes comme **Procédure** seront expliqués un peu plus loin. Dans cet exemple, vingt tirets sont affichés, en partant du principe qu'ils le sont sur la même ligne.

```
Procédure RepeteCar()
Var
i:entier
Début
  Pour i de 1 à 20 Faire
    Afficher "-"
  FinPour
FinProc
```

Vous constatez qu'un sous-programme est constitué de :

- Un identifiant sous forme de nom : RepeteCar(), qui lui servira pour être appelé.
- Une zone de déclaration de variables.
- Un bloc d'instructions encadré entre Début et Fin.
- Le tout étant ici et dans le cadre de cet exemple, précisément entre les mots-clés Procédure et FinProc.

En algorithmique, un sous-programme ne peut pas être déclaré dans un autre sous-programme. La possibilité existe cependant parfois dans certains langages de programmation, comme le PHP, mais alors, il faut faire preuve d'une très grande rigueur : le sous-programme sera connu du reste du programme seulement quand celui qui le déclare sera lui-même appelé une première fois.

b. En PHP

En PHP, un sous-programme se déclare sous cette forme :

```
function nom_fonction(argument1, argument2, ..., argumentn) {
  /* code du sous programme */
  return valeur ; /* suivant le retour */
}
```

Vous n'avez pas encore rencontré ce qu'est un retour ni un argument. Ce n'est pas gênant pour les petits exemples.

Voici le petit sous-programme RepeteCar() en PHP.

```
  function RepeteCar() {
    for($i=1 ;$i<=20 ;$i++) echo "*";
    echo "<br />";
  }
```

3. Appel

Un sous-programme est exécuté depuis le programme principal ou un autre programme. Pour cela, le programme fait appel au sous-programme. L'appel au sous-programme est une instruction qui va déclencher l'exécution de celui-ci. Cet appel peut avoir lieu n'importe où. Suivant les diverses conventions, les sous-programmes peuvent être appelés depuis une instruction **Appel**, ou **Appeler**. Plus simplement, il est d'usage d'appeler un sous-programme par son nom. L'usage peut cependant changer selon le type de sous-programme, procédure ou fonction, les instructions ci-dessus étant souvent réservées aux procédures. Pour reprendre le sous-programme RepeteCar(), voici un exemple d'appel :

```
Procédure RepeteCar()
VAR
i:entier
DEBUT
  Pour i de 1 à 20 Faire
    Afficher "-"
  FinPour
FIN
FinProc
PROGRAMME LIGNES
VAR
  i:entier
DEBUT
  Afficher "Voici 10 lignes de 20 caractères"
  Pour i de 1 à 10 Faire
    RepeteCar()
  FinPour
  Afficher "Le programme est terminé"
FIN
```

Les plus perspicaces d'entre vous auront remarqué que la variable i est déclarée deux fois, une fois dans le sous-programme et une fois dans le programme principal. C'est normal, les explications arriveront en temps voulu.

Lorsque le sous-programme se termine, l'instruction située juste en dessous de son appel est exécutée. On dit qu'il y a retour du sous-programme vers l'instruction suivante, ou appelante.

Enfin, un sous-programme peut être appelé en lui fournissant des valeurs, appelées paramètres. Ces valeurs seront placées dans des variables utilisables au sein du sous-programme comme toute autre variable.

Voici l'exemple complet en PHP avec en gras l'appel à la méthode (sous-programme).

```
<html>
  <head><meta/>
    <title>déclaration de fonction</title>
  </head>
  <body>
  <?php
  function RepeteCar() {
    for($i=1 ;$i<=20 ;$i++) echo "*";
    echo "<br />";
  }

  echo "Affichage de 10 lignes de 20 caractères<br />";
  for($i=1;$i<=10;$i++) RepeteCar();
?>
  </body>
</html>
```

4. Fonctions et procédures

Le sous-programme RepeteCar() a été déclaré avec le mot-clé **Procédure** et il vous a été indiqué aussi l'existence du mot-clé **Fonction**. Il y a donc deux types de sous-programmes. Certains langages peuvent proposer ou l'un, ou l'autre, parfois les deux. Vous entendrez parfois parler de langages procéduraux (comme le Pascal) ou fonctionnels (comme le C).

Avant de continuer, juste un petit mot sur PHP. PHP ne fait pas de différences entre les fonctions et les procédures telles que présentées ici. En PHP, les procédures sont des fonctions qui ne retournent pas de valeurs.

a. Les procédures

Les procédures sont des sous-programmes constitués d'une suite d'instructions indépendantes. Une procédure ne retourne pas de résultat ou de valeur au programme qui l'a appelé, tout comme les valeurs passées en paramètre ne sont pas forcément modifiées de manière globale. Une procédure pourrait faire l'objet d'un programme à part. Son contenu peut cependant parfois influer sur le déroulement global du programme, s'il modifie un fichier, une base de données, etc.

La procédure RepeteCar() est un exemple typique : le bloc de données répétitif influe sur l'affichage, mais ne retourne rien en tant que tel.

Il existe cependant quelques moyens indirects pour une procédure de retourner une valeur :

- en passant celle-ci par référence, selon le même principe que ce qui a été expliqué pour les tableaux : le sous-programme reçoit la référence de la variable et peut la modifier vers celle contenant la nouvelle valeur. Ceci est d'ailleurs aussi possible avec les fonctions.
- Plus simplement, l'algorithmique effectue souvent une distinction entre les valeurs en entrée de la procédure (celles qu'on lui transmet) et les valeurs en sortie. Dans ce cas, autant utiliser les fonctions, notamment si une seule valeur doit être retournée.
- En modifiant les contenus des variables globales, variables accessibles pour tous les programmes et les sous-programmes tant en lecture qu'en écriture (cf. section "Variables locales et globales" de ce chapitre).

b. Les fonctions

En mathématique, vous avez probablement rencontré la notion de fonction. Dans le cadre de la résolution d'une équation du second degré, l'équation s'écrit généralement ainsi :

```
f(x)=ax²+bx+c
```

Le résultat de f(x) ou fonction de x est le résultat du contenu de la fonction, c'est-à-dire l'équation. La valeur de f(x) est ce résultat. C'est pareil en algorithmique : une fonction est un sous-programme qui retourne une valeur. Un fonction se décrit ainsi :

```
Fonction nom():type
Var
  /* variables */
Début
  /* bloc d'instructions */
  Retourne valeur
FinFonc
```

- Une fonction se déclare avec le mot-clé **Fonction**, suivi de son nom et du type de valeur qu'elle retourne. Ce peut être n'importe quel type (entier, réel, chaîne…).
- Une fonction peut contenir une zone de déclaration de variables et de types structurés.
- Le bloc d'instructions est encadré par **Début** et **FinFonc**.
- La valeur de la fonction est retournée par l'instruction **Retourne**. La valeur retournée doit être du même type que celle attendue par la déclaration de la fonction.

```
Fonction equation():réel
Var
  a,b,c,x:réels
Début
  x←a*x*x+b*x+c
  retourne x
FinFonc
```

Il existe une différence très importante entre une procédure et une fonction :

- La procédure est vue comme une **instruction**.
- La fonction est vue comme une **valeur**.

Tout comme une variable retourne une valeur, une fonction retourne aussi une valeur, ce qui veut dire qu'une fonction peut être utilisée (appelée) partout où une variable pourrait l'être : dans une expression, dans un calcul, un affichage, une affectation, etc. Dans un seul cas, la fonction ne peut pas être utilisée : une fonction fournit une valeur, elle ne peut pas se voir affectée une valeur. Ceci est interdit :

```
equation←x ; /* INTERDIT */
```

mais ceci est autorisé :

```
x←equation()
```

x recevra alors la valeur retournée par la fonction equation() via l'instruction **Retourne**.

```
PROGRAMME EQ1
Var
  result:réel
Début
  result←equation()
  Afficher result
Fin
```

L'instruction **Retourne** ne retourne pas une variable, mais une valeur. Cette valeur peut être le contenu d'une variable, mais aussi une autre fonction (auquel cas c'est le résultat de cette autre fonction qui sera retourné) ou n'importe quelle expression pouvant être évaluée. La fonction equation() peut donc être écrite ainsi :

```
Fonction equation():réel
Var
  a,b,c,x:réels
Début
  Retourne a*x*x+b*x+c
FinFonc
```

Voici le même exemple en PHP. Cette fois la méthode va retourner un entier, ce qui est précisé lors de sa déclaration.

```
function equation() {

  /* il faudrait initialiser les variables ici */
  $x=$a*$x*$x+$b*$x+$c ;
  return $x ;
}
```

5. Variables locales et globales

a. Locales

L'exemple de la procédure RepeteCar() a soulevé un petit problème pas si anodin que cela. Il met en évidence l'utilisation de deux variables de même nom, l'une dans le programme principal, l'autre dans le sous-programme. La variable i apparaît deux fois.

L'endroit où les variables sont déclarées est très important. Selon cet endroit, les variables ont une "portée" différente. La portée d'une variable est sa visibilité au sein des différentes parties du programme.

Le cas général dit qu'une variable n'est visible et accessible par défaut que dans le bloc d'instructions où elle a été déclarée. Une variable déclarée dans un sous-programme sous les mots-clés **Procédure** ou **Fonction** ne pourra dans ce cas qu'être lisible et modifiable uniquement dans ce sous-programme. Idem pour le programme principal : une variable déclarée sous le mot-clé **Programme** ne sera accessible que par celui-ci.

Les variables accessibles uniquement par le programme ou sous-programme dans lesquels elles sont déclarées, sont appelées des variables locales. Toutes les variables que vous avez rencontrées jusqu'à présent sont des variables locales.

Les variables locales de même nom n'ont aucun rapport entre elles. Elles sont totalement indépendantes les unes

des autres et aucune interaction entre elles n'est possible. Les variables locales peuvent donc parfaitement porter un même nom. La variable i de RepeteCar() n'est pas du tout la même que la variable i du programme Lignes. Il n'y a aucun risque d'accéder à la valeur ou de modifier celle-ci par accident de l'un vers l'autre programme. Du côté de la mémoire, le contenu de ces deux variables est cloisonné et distinct, à des adresses différentes.



En PHP, une variable déclarée dans le programme principal ou dans une méthode est locale au programme principal, c'est-à-dire non visible depuis les autres fonctions, sauf si c'est explicitement précisé.

b. Globales

Il serait pourtant très pratique de pouvoir accéder à une variable depuis n'importe quel endroit du programme, qu'il soit principal ou un sous-programme. Ce mécanisme permettrait d'utiliser son contenu et d'en modifier la valeur partout dans le programme. La portée d'une telle variable s'étendrait à tout le code. Ce type de variable s'appelle une variable globale et elle existe tant en algorithmique que dans la plupart des langages.

Une variable globale est déclarée en dehors des sous-programmes et du programme principal, avant ceux-ci, c'est-à-dire en premier dans l'algorithme. Étant globale, elle est accessible de partout, tant en accès (lecture du contenu) qu'en modification (affectation d'une nouvelle valeur). Les variables globales sont déclarées de cette manière :

```
Var globales
  nbcar:entier
  c:caractère
Procédure RepeteCar()
VAR
i:entier
Début
  Pour i de 1 à nbcar Faire
    Afficher c
  FinPour
FinProc

PROGRAMME LIGNES
VAR
  i:entier
DEBUT
  c←"*"
  Pour nbcar de 1 à 10 Faire
    RepeteCar()
  FinPour
FIN
```

Ce programme une fois exécuté devrait vous afficher quelque chose de ce genre :

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Remarquez que les deux variables globales sont modifiées dans le programme principal, tandis que le sous-programme y accède.

La variable globale amène trois remarques :

- Elle n'est déclarée qu'une seule fois pour l'intégralité du programme. Elle ne doit donc jamais être redéclarée tant dans le programme principal que dans un sous-programme.
- Elle permet indirectement de "passer" des valeurs aux sous-programmes qui l'utilisent. C'est un "dommage collatéral" et les variables globales ne devraient être réservées que lorsque celles-ci sont vraiment communes à la plus grande partie du code.

- Il serait ridicule de déclarer toutes les variables en globales. La plupart des programmes et sous-programmes ne les utiliseraient pas toutes et en plus vous risqueriez par accident, pensant à une variable locale, d'en modifier certaines sans y prendre garde et ainsi de mettre en péril l'exécution du programme.

- Comme corollaire, ne donnez jamais le même nom à une variable locale et globale. C'est interdit en algorithmique et dans la plupart des langages qui ne manqueront pas de vous le faire remarquer à la compilation ou à l'exécution.

c. Variables globales et PHP

En PHP les variables ne sont pas globales par défaut. Si vous souhaitez accéder et modifier une variable du bloc principal au sein d'une fonction vous devez l'indiquer explicitement au sein de la fonction en précédant la variable du mot-clé **global**.

```
<html>
  <head><meta/>
    <title>portee des variables</title>
  </head>
  <body>
  <?php
  function RepeteCar() {
    global $c;
    for($i=1 ;$i<=20 ;$i++) echo "$c";
    echo "<br />";
  }
  $c='*';
  for($nbcar=1;$nbcar<=10;$nbcar++) RepeteCar();
?>
  </body>
</html>
```

6. Les paramètres

Maintenant que vous réservez les variables globales à des cas bien précis, il vous faut trouver un autre moyen de passer des valeurs aux procédures et aux fonctions. Quand en mathématiques vous calculez la valeur d'une fonction f (x), vous lui demandez en fait de calculer la valeur de la fonction selon la valeur de x. Vous passez donc la valeur de x à la fonction.

Le principe est le même avec vos algorithmes : vous pouvez transmettre des valeurs à vos procédures et fonctions, tout comme vous pouvez en recevoir. La syntaxe diffère légèrement entre les fonctions et les procédures pour cette raison. Dans les deux cas cependant, les paramètres se placent entre les parenthèses situées après leur nom.

Les paramètres passés à un sous-programme sont généralement des variables locales au programme ou sous-programme l'appelant, mais ils ne portent pas forcément le même nom. Ils sont "récupérés" au sein du sous-programme comme des variables locales au sous-programme. Il est cependant possible de passer comme paramètre toute expression retournant une valeur, que ce soit un scalaire, une variable, un tableau, un enregistrement, une table, ou encore une fonction (qui sera substituée par son résultat).

a. Procédures

La syntaxe de passage des paramètres est la suivante :

```
Procédure nom_proc(E param1:type,ES param2:type,S param3:type)
```

Les paramètres d'une procédure sont de trois catégories :

- Ceux en entrée, qui correspondent aux valeurs que vous souhaitez transmettre à la procédure. Entre les parenthèses, ils sont précédés d'un "**E**", comme Entrée, car ce sont les valeurs en entrée de la procédure.

- Ceux en sortie, qui correspondent aux valeurs retournées par la procédure au programme ou au sous-programme l'ayant appelé. Ils sont précédés d'un "**S**", comme Sortie, car ce sont les valeurs en sortie de la procédure. Ces paramètres sont des variables qui doivent être déclarées dans le programme ou sous-programme appelant.

- Ceux en entrée et en sortie, précédés de "**ES**".

Quand vous avez plusieurs paramètres en entrée, il suffit de tous les mettre après le E et même de les regrouper selon leur type. Cette procédure prend cinq paramètres : trois entiers et deux chaînes.

```
Procédure proc(E p1,p2,p3:entiers,p4,p5:chaînes)
```

La procédure RepeteCar() se prête bien à un paramètre. Comment en effet utiliser une variable globale pour spécifier le nombre de caractères à répéter ? Autant passer ce nombre en paramètre. Voici le programme modifié en conséquence :

```
Procédure RepeteCar(E nbcar:entier, E c:caractère)
VAR
i:entier
DEBUT
  Pour i de 1 à nbcar Faire
    Afficher c
  FinPour
FinProc

PROGRAMME LIGNES
VAR
  i:entier
DEBUT
  Pour i de 1 à 10 Faire
    RepeteCar(i,"*")
  FinPour
FIN
```

Vous récupérez une éventuelle valeur en sortie via la ou les variables E ou ES. Voici une procédure qui convertit un nombre de secondes en heures, minutes et secondes. Elle prend quatre paramètres dont un en entrée (le nombre de secondes) et trois en sortie (heures minutes et secondes).

```
Procédure sec_to_hms(E nbsec:entier, S h,m,s:entiers)
Début
  h←nbsec DIV 3600
  nbsec←nbsec%3600
  m←nbsec DIV 60
  s←nbsec MOD 60
FinProcédure

PROGRAMME convert_sec
Var
  nb_secondes :entier
  heures,minutes,secondes:entiers
Début
  nb_secondes←3950
  sec_to_hms(nb_secondes,heures,minutes,secondes)
  Afficher heures,minutes,secondes
Fin
```

Passer un enregistrement comme paramètre

La procédure précédente fonctionne certes à merveille, mais avec un peu plus de réflexion, pourquoi ne pas utiliser une seule structure pour représenter toutes les composantes d'une heure ?

```
Struct hms
  heures:entier
  minutes:entier
  secondes:entier
FinStruct
```

Un enregistrement peut être passé en paramètre d'une procédure tout comme une variable, en entrée ou en sortie. La procédure et le programme peuvent être convertis comme ceci :

```
Types
Struct hms
```

```
  heures:entier
  minutes:entier
  secondes:entier
FinStruct

Procédure sec_to_hms(E nbsec:entier, S duree:hms)
Début
  duree.heures←nbsec DIV 3600
  nbsec←nbsec%3600
  duree.minutes←nbsec DIV 60
  duree.secondes←nbsec%60
FinProcédure

PROGRAMME convert_sec
Var
  nb_secondes:entier
  heures:hms
Début
  nb_secondes←3950
  sec_to_hms(nb_secondes,heures)
  Afficher heures.heures,heures.minutes,heures.secondes
Fin
```

b. Les fonctions

Les fonctions ne retournent qu'une seule valeur via l'instruction Retourne. Aussi, il n'y a pas besoin de spécifier si les paramètres sont en entrée ou en sortie. Ils sont forcément en entrée. Par contre, la valeur de la fonction peut être de n'importe quel type.

Voici une fonction qui fait l'inverse de la procédure précédente : elle reçoit des heures, minutes et seconde et en contrepartie elle retourne le nombre de secondes total.

```
Fonction hms_to_sec(heures,minutes,secondes :entiers):entier
Var
  total:entier
Début
  total←heures*3600+minutes*60+secondes
  Retourne total
FinFonc
```

Une fonction peut parfaitement récupérer un enregistrement comme paramètre. Voici une bonne occasion de réutiliser le type structuré précédemment défini :

```
Fonction hms_to_sec(duree:hms):entier
Var
  total :entier
Début
  total←duree.heures*3600+duree.minutes*60+duree.secondes
  Retourne total
FinFonc
```

Enfin, une fonction peut aussi retourner un enregistrement. Ce qui veut dire que la procédure `sec_to_hms` est inutile : une fonction peut la remplacer :

```
Fonction sec_to_hms(nbsec :entier):hms
Var
  duree:hms
Début
  duree.heures←nbsec DIV 3600
  nbsec←nbsec%3600
  duree.minutes←nbsec DIV 60
  duree.secondes←nbsec%60
  Retourne duree
FinFonc
```

Pour cette dernière fonction, le résultat doit être affecté à un enregistrement de même type.

```
PROGRAMME convert_sec2
```

```
Var
  nb_secondes:entier
  heures:hms
Début
  nb_secondes←3950
  heures←sec_to_hms(nb_secondes)
  Afficher heures.heures,heures.minutes,heures.secondes
Fin
```

Une procédure ne retournant qu'une seule valeur et quel que soit son type est toujours convertible en fonction. Une telle procédure doit d'ailleurs toujours être convertie en ce sens : elle n'a pas d'intérêt autrement.

Pour tester les fonctions `sec_to_hms` et `hms_to_sec`, vous pouvez appeler l'une avec l'autre. Une fonction représentant son résultat (la valeur de la fonction est le résultat qu'elle retourne), elle peut être utilisée comme paramètre à une autre fonction si les types attendus sont compatibles.

```
PROGRAMME convert_sec3
Var
  nb_secondes1:entier
  nb_secondes2:entier
Début
  nb_secondes1←3950
  nb_secondes2←hms_to_sec(sec_to_hms(nb_secondes1))
  Si nb_secondes1!=nb_secondes2 Alors
    Afficher "Erreur dans l'une des fonctions ?"
  Sinon
    Afficher "Un programme parfait"
  FinSi
Fin
```

c. Paramètres et PHP

Le langage PHP accepte n'importe quel type de paramètre en argument d'une fonction : types classiques comme les entiers, ou tableaux, structures (enregistrements), objets. Dans certains cas, ces paramètres peuvent être en entrée et en sortie, quand ils sont passés en tant que référence ou si ce sont des objets. Les références seront abordées un peu plus profondément dans le chapitre consacré aux notions avancées dont les pointeurs.

En attendant, tous les exemples précédents sont réalisables en PHP, bien heureusement. La procédure RepeteCar () est donc transformée ainsi :

```
<html>
  <head><meta/>
    <title>Parametres</title>
  </head>
  <body>
  <?php
  function RepeteCar($nbcar, $c) {
    for($i=1 ;$i<=$nbcar ;$i++) echo "$c";
    echo "<br />";
  }
  for($nb=1;$nb<=10;$nb++) RepeteCar($nb,'*');
?>
  </body>
</html>
```

Voici en PHP une reprise des structures hms et des fonctions `hms_to_sec()` et `sec_to_hms()`. Il n'y a pas de difficulté dans cette implémentation :

```
<html>
  <head><meta/>
    <title>fonctions hms</title>
  </head>
  <body>
  <?php
```

```
  class hms {
    public $heures;
    public $minutes;
    public $secondes;
}

  function hms_to_sec($duree) {
    $total=$duree->heures*3600+$duree->minutes*60+$duree->secondes;
    return $total;
  }

  function sec_to_hms($nbsec) {
    $duree=new hms;
    $duree->heures=(int)($nbsec/3600);
    $nbsec=$nbsec%3600;
    $duree->minutes=(int)($nbsec/60);
    $duree->secondes=$nbsec%60;

    return $duree;
  }

  // exemple 1
  $nb_secondes=39450;
  $heures=sec_to_hms($nb_secondes);
  echo $heures->heures.":".$heures->minutes.":".$heures->secondes."<br />";

  // exemple 2
  $nb_secondes1=3950;
  $nb_secondes2=hms_to_sec(sec_to_hms($nb_secondes1));
  echo "$nb_secondes2<br />";
?>
  </body>
</html>
```

d. Petite application fonctionnelle

Pour finir, ces deux fonctions sont plus pratiques qu'il n'y paraît. Grâce à elles, vous pouvez très facilement trouver le temps écoulé entre une première heure et une seconde heure. Le principe est des plus simples :

- Vous saisissez deux horaires dans des enregistrements hms.
- Vous les convertissez en nombre de secondes.
- Vous faites la différence entre ces deux nombres.
- Vous convertissez le résultat en enregistrement hms.

Dit comme ceci, cela pourrait sembler un peu compliqué si les fonctions n'existaient pas. Vous avez donc besoin de la définition du type structuré hms et des deux fonctions `nbsec_to_hms` et `hms_to_nbsec`. Vous n'avez même pas besoin de variables pour y placer les nombres de secondes. Une seule ligne avec les bons paramètres suffit à s'en passer (voyez celle en gras).

```
Type
Struct hms
  heures:entier
  minutes:entier
  secondes:entier
FinStruct

Fonction hms_to_sec(duree:hms):entier
Var
  total:entier
Début
  total←duree.heures*3600+duree.minutes*60+duree.secondes
  Retourne total
FinFonc
```

```
Fonction sec_to_hms(nbsec :entier):hms
Var
  duree:hms
Début
  duree.heures←nbsec DIV 3600
  nbsec←nbsec%3600
  duree.minutes←nbsec DIV 60
  duree.secondes←nbsec%60
  Retourne duree
FinFonc

Programme delta
Var
  heure1,heure2,delta:hms
Début
  Afficher "Saisir l'heure de départ"
  Saisir heure1.heures,heure1.minutes,heure1.secondes
  Afficher "Saisir l'heure de fin"
  Saisir heure2.heures,heure2.minutes,heure2.secondes
  delta←sec_to_hms(hms_to_nbsec(heure2)-hms_to_nbsec(heure1))
  Afficher "Il s'est écoulé :"
  Afficher delta.heures,delta.minutes,delta.secondes
Fin
```

Pourquoi ne pas aller encore plus loin en créant une fonction qui calcule seule la durée en faisant appel aux deux autres fonctions ? En effet, vous pouvez parfaitement appeler des sous-programmes dans d'autres sous-programmes, donc des fonctions dans des fonctions, ou des procédures dans des fonctions et réciproquement. Voici une fonction `hms_delta` qui va calculer tout ceci :

```
Fonction hms_delta(hms1,hms2:hms):hms
Var
  delta:hms
Début
  delta←nbsec_to_hms(hms_to_nbsec(hms2)-hms_to_nbsec(hms1))
  Retourne delta
FinFonc
```

Le programme principal est modifié comme ceci :

```
Programme delta
Var
  heure1,heure2,delta:hms
Début
  Afficher "Saisir l'heure de départ"
  Saisir heure1.heures,heure1.minutes,heure1.secondes
  Afficher "Saisir l'heure de fin"
  Saisir heure2.heures,heure2.minutes,heure2.secondes
  delta←hms_delta(heure1,heure2)
  Afficher "Il s'est écoulé :"
  Afficher delta.heures,delta.minutes,delta.secondes
Fin
```

Voici enfin une implémentation en PHP de la fonction `hms_delta()` et du calcul de ce dernier exemple.

```
...
  function hms_delta($hms1, $hms2) {
    $delta=new hms;
    $delta=sec_to_hms(hms_to_sec($hms2)-hms_to_sec($hms1));
    return $delta;
  }
...

  // exemple 3
  $heures->heures=10;
  $heures->minutes=30;
  $heures->secondes=45;
  $heures2->heures=18;
  $heures2->minutes=36;
```

```
  $heures2->secondes=24;

  $delta=hms_delta($heures,$heures2);
  echo $delta->heures.":".$delta->minutes.":".$delta->
secondes."<br />";
...
```

7. Sous-programmes prédéfinis

a. Un choix important

Presque tous les langages de programmation proposent d'une manière ou d'une autre la possibilité de créer des sous-programmes. C'est le cas du C, du C++, de Java, de C#, PHP, etc. Vous pouvez et vous allez créer vos propres sous-programmes, selon le cas.

Cependant, ces mêmes langages sont souvent déjà fournis avec une quantité plus ou moins importante de sous-programmes. Dans les exemples précédents, vous en avez déjà utilisés. En PHP, les instructions **count** ou **fgets** sont des sous-programmes, l'un chargé de compter le nombre d'éléments dans un tableau, l'autre de saisir des données (ou d'en lire dans un fichier). Le langage PHP propose une quantité impressionnante de sous-programmes : calculs mathématiques, travaux graphiques, manipulations de bases de données : plusieurs milliers !

Autrement dit, les concepteurs du langage ou d'autres personnes fournissent des bibliothèques de sous-programmes pour réduire autant que possible et sans tomber dans l'excès, le travail des programmeurs. Avant de créer vos propres sous-programmes, vérifiez tout d'abord dans la documentation du langage que vous utilisez, si vous pouvez y trouver votre bonheur.

Un sous-programme prédéfini porte un nom qui reflète souvent ce qu'il fait. C'est une procédure ou une fonction (cf. section "Variables locales et globales" de ce chapitre), plus souvent une fonction, qui retourne un résultat.

b. Quelques exemples

Fonctions sur chaînes

Vous pouvez obtenir la longueur d'une chaîne de caractères avec la fonction **longueur**. Elle s'emploie en passant entre parenthèses une chaîne de caractères et retourne comme résultat une valeur entière qui est le nombre de caractères de cette chaîne.

```
Programme len
Var
  txt :chaîne
  l :entier
Début
  txt←"Un petit texte"
  l←longueur(txt)
  Afficher l /* affiche 14 */
Fin
```

Les fonctions **milieu**, **gauche** et **droite** permettent de découper des morceaux dans une chaîne de caractères. La fonction **milieu** prend trois valeurs entre parenthèses : une chaîne, une position de début et une longueur. Elle est parfois nommée **sschaine** (sous-chaîne), s'utilisant de la même manière. Les deux autres ne prennent que deux paramètres : une chaîne et une longueur. Dans le cas de gauche, c'est le nombre de caractères à découper en partant de la gauche, pour droite depuis la droite et pour milieu depuis la position indiquée. Les positions démarrent à 1. Le programme suivant va tout d'abord afficher tous les caractères d'une chaîne, les uns après les autres, puis le premier mot, puis le dernier.

```
Programme decoupe
Var
  txt :chaîne
  result :chaîne
  i :entier
Début
  txt="Salut les amis"
  Pour i de 1 à longueur(txt) Faire
    result←milieu(txt,i,1)
    Afficher result // les lettres une à une
  Fin
```

```
  Affiche gauche(txt,5) // Salut
  Affiche droite(txt,4) // amis
Fin
```

La fonction **pos** (position) détermine la position d'une chaîne de caractères dans une autre. Elle trouve donc une sous-chaîne de caractères donnée et retourne sa position le cas échéant, zéro sinon.

```
Programme trouve
Var
  ch:chaînes
  position :entier
Début
  txt←"abcedefghik"
  position←pos("def",txt)
  Si pos=0 Alors
    Afficher "Pas trouvé"
  sinon
    Afficher "A la position",pos
  FinSI
Fin
```

La fonction **suppr** permet de supprimer une sous-chaîne d'une chaîne de caractères en fonction de sa position initiale et de sa longueur.

```
Programme suppr
Var
  txt :chaîne
Début
  txt←"abcdefgh"
  txt←suppr(txt,4,len(txt)-3)
  Affiche txt /* reste abc */
Fin
```

Fonctions mathématiques

Tous les langages proposent souvent un grand nombre de fonctions mathématiques. Dommage pour vous, la plupart des exemples des derniers chapitres ont des équivalents sous forme de fonctions. C'est le cas des racines carrées, des puissances, des factorielles, des fonctions trigonométriques comme les sinus, cosinus et tangente, etc.

- racine(x) : donne la racine carrée de x
- puissance(x,y) : donne x à la puissance y
- sin(x) : sinus de x
- cos(x) : cosinus de x
- tan(x) : tangente de x

Certaines conventions algorithmiques ne reconnaissant pas les % ou MOD comme opérateurs modulos (alors que les langages, eux, si), vous pouvez trouver parfois la fonction **mod**(x,y), qui équivaut à x MOD y et la fonction **entier** (x,y) qui équivaut à x DIV y.

Une fonction très sympathique est la fonction **aléatoire()**. Elle détermine un nombre aléatoire. Utilisée sans paramètre, elle trouve un réel entre 0 et 1. Utilisée avec un paramètre entier, elle trouve une valeur entre 0 et n. Idéal pour un lancer de dé.

```
Programme dé
Var
  dé, saisie :entiers
Début
  dé←aléatoire(5)+1 // entre 1 et 6
  Répéter
    Afficher "Quelle est la valeur du dé ?"
    Saisir saisie
    Si dé<>saisie Alors
```

```
      Afficher "Raté, essayez encore"
    FinSi
  Jusqu'à dé=saisie
  Afficher "Bravo !"
Fin
```

Fonctions de conversion

Vous pouvez convertir une chaîne de caractères en valeur numérique à l'aide de la fonction **chnum**, qui prend comme paramètre la chaîne de caractères et qui retourne la valeur en entier ou réel (selon le cas). La chaîne doit être une représentation d'une valeur numérique et rien d'autre.

La fonction **numch** fait exactement le contraire : elle convertit un nombre en chaîne de caractères, l'idéal pour sauver tout ceci dans un fichier texte. Voici un exemple simple :

```
Programme conversion
Var
  rPi :réel
  cPi :chaîne
Début
  rPi=3.1415927
  cPi=numch(rPi)
  Afficher cPi
  cPi="3.14"
  rPi=chnum(cPi)
Fin
```

Fonctions prédéfinies en PHP

Les fonctions prédéfinies en PHP sont extrêmement nombreuses. Dans les exemples précédents vous avez eu l'occasion d'en rencontrer quelques-unes, notamment des conversions de chaînes vers des entiers, par exemple. Les fonctions prédéfinies sont en fait souvent associées à des types particuliers, à des objets particuliers, etc. Le mieux est de vous reporter à la documentation en ligne de PHP à l'adresse http://www.php.net/manual/fr/. Voici quelques courts exemples qui peuvent vous intéresser :

- **strlen()** : retourne la longueur de la chaîne.
- **substr(chain, début [,fin])** : découpe un morceau de chaîne.
- **trim()** : supprime tous les espaces en début et fin de chaîne.
- **isempty()** : retourne vrai ou faux selon que la chaîne est vide ou non.

Par exemple :

```
<html>
  <head><meta/>
    <title>fonctions internes</title>
  </head>
  <body>
  <?php

  $txt1="  Bonjour les amis  ";
  $txt1=trim($txt1);
  $valeur=strlen($txt1);
  echo "$txt1 de longueur $valeur<br />";

  $txt2=substr($txt1,3,5);
  echo "$txt2<br />";
?>
  </body>
</html>
```

8. Dernier cas : les tableaux

Une fonction peut retourner un tableau seulement si la variable qui la reçoit est elle-même un tableau de même dimension et de même nombre d'indices.

De même, il est possible de passer un tableau en paramètre d'une fonction. Dans ce cas, vous ne connaissez pas forcément par avance le nombre d'éléments du tableau, aussi vous pouvez ne rien préciser entre les crochets. Cependant, si le langage d'implémentation ne propose pas d'instructions ou de fonctions prédéfinies pour trouver la taille d'un tableau, vous devriez passer celle-ci en paramètre. Quand vous passez un tableau en paramètre d'un sous-programme, mettez uniquement son nom sans utiliser les crochets.

De nombreux exemples sont possibles. Parmi eux, pourquoi ne pas trier un tableau ? Reprenez l'un des algorithmes de tri du chapitre précédent, comme le tri par insertion et adaptez-le pour le transformer en fonction :

```
Fonction tri_tableau(t:tableau[] d'entiers,nb:entier) :tableau[] d'entiers
VAR
  i,mem:entiers
DEBUT
  Pour i de 1 à nbelem faire
    mem←tab[i]
    pos←i-1
    tant que tab[pos]>mem et pos>=0 Faire
      tab[pos+1]←tab[pos]
      pos←pos-1

    FinTantQue
    tab[pos+1]←mem
FinPour
FinFonc
```

Voici un petit programme pour l'exploiter :

```
PROGRAMME TRITAB
CONST
  INDICES=10
VAR
  t :tableau[1..INDICES] ←{10,5,8,7,3,1,6,9,4,2} d'entiers
  i :entier
Début
  t=tri_tableau(t,INDICES)
  Pour i de 1 à INDICES Faire
    Afficher t[i]
  FinPour
Fin
```

En PHP c'est encore plus simple : la taille des tableaux passés en paramètres n'a pas d'importance. Vous pouvez facilement obtenir la taille d'un tableau avec la fonction count(). La fonction `tri_tableau()` ne reçoit donc qu'un seul paramètre, le tableau à trier et n'est même pas obligée de retourner le tableau car il est passé en référence.

```
<html>
  <head><meta/>
    <title>Fonction de tri de tableau</title>
  </head>
  <body>
  <?php

  // attention passage par référence
  function tri_tableau(&$tab) {

    $cpt=count($tab);

    for($i=1;$i<$cpt;$i++) {
      $mem=$tab[$i];
      $pos=$i-1;
      while(($pos>=0) && ($tab[$pos]>$mem)) {
        $tab[$pos+1]=$tab[$pos];
        $pos--;
      }
      $tab[$pos+1]=$mem;
    }
  }
```

```
  $t=array(48,17,25,9,34);
  $cpt=count($t);

  echo "Avant :";
  for($i=0;$i<$cpt;$i++) echo $t[$i]." ";
  echo "<br />";

  tri_tableau($t);

  echo "Apres :";
  for($i=0;$i<$cpt;$i++) echo $t[$i]." ";
  echo "<br />";

?>
  </body>
</html>
```

Le résultat est le suivant :

```
Avant :
48 17 25 9 34
Après :
9 17 25 34 48
```

Les sous-programmes récursifs

1. Principe

Un sous-programme peut appeler un autre sous-programme, quel qu'il soit. Donc un sous-programme peut s'appeler lui-même. Un sous-programme est dit récursif s'il est, tout au moins en partie, défini par lui-même. Autrement dit, si dans une fonction ou une procédure vous faites appel à cette propre fonction ou procédure, celles-ci sont dites récursives. L'exemple le plus simple est la factorielle : **n!**=n***(n-1)!**

Il existe deux types de récursivité :

- Simple ou rapide : le sous-programme s'appelle lui-même.
- Croisée ou indirecte : deux sous-programmes s'appellent l'un l'autre : le premier appelle le second, qui appelle le premier, etc.

La récursivité peut être appliquée tant aux fonctions qu'aux procédures.

Pour une récursivité simple :

```
Procédure recursive()
Début
  /* instructions */
  recursive()
  /* instructions */
Fin
```

Pour une récursivité croisée :

```
Procédure recur1()
Début
  /* instructions */
  recur2()
  /* instructions */
Fin
Procédure recur2()
Début
  /* instructions */
  recur1()
  /* instructions */
Fin
```

La suite ne va exposer que les sous-programmes récursifs simples.

2. Un premier exemple : la factorielle

Une factorielle est l'exemple rêvé d'application d'un algorithme récursif. Cet exemple a déjà été présenté dans les chapitres précédents mais un petit rappel s'impose :

- 10!=10*9*8*7*6*5*4*3*2*1
- Donc 10!=10*(9*8*7*6*5*4*3*2*1)
- Donc 10!=10*9!
- Donc n!=n*(n-1)!

Si vous créez une fonction (appropriée dans ce cas) appelée fact() et chargée de calculer la factorielle de n, vous auriez un raccourci de ce genre :

```
fact(n)=n*fact(n-1)
```

De là, il vous devient très facile d'écrire une fonction fact() récursive :

```
Fonction fact(n:entier) :entier
Début
  n←fact(n-1)
  Retourne n
Fin
```

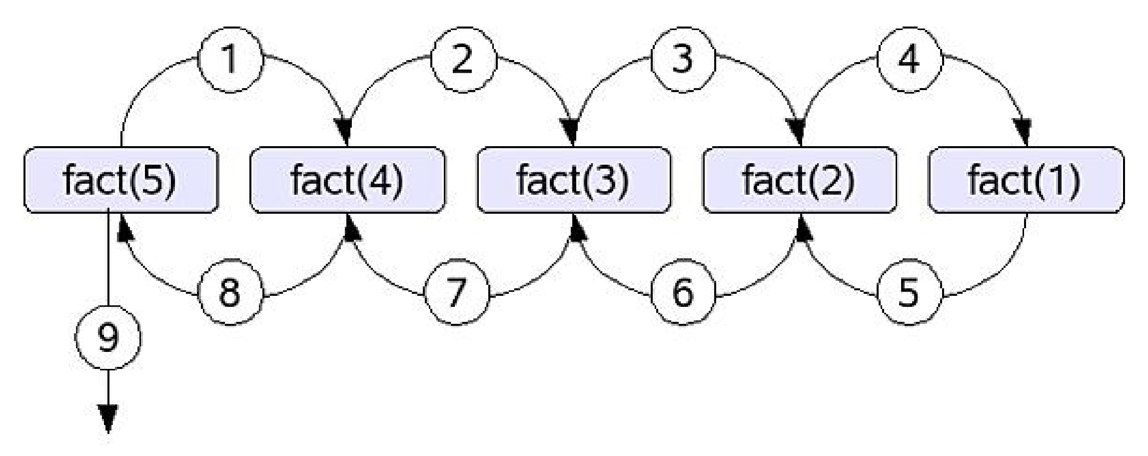
Cette fonction n'est pas complète car elle va s'exécuter à l'infini. Il n'y a pas de condition d'arrêt. Or, le calcul doit continuer tant que n est supérieur à 1. Voici les passes successives pour une factorielle de 5 :

- 1ère étape : 5>1 ? Oui : fact(5) appelle 5*fact(4)
- 2ème étape : 4>1 ? Oui : fact(4) appelle 4*fact(3)
- 3ème étape : 3>1 ? Oui : fact(3) appelle 3*fact(2)
- 4ème étape : 2>1 ? Oui : fact(2) appelle 2*fact(1)
- 5ème étape : 1>1 ? Non : fact(1) sort en retournant la valeur 1 à fact(2).

Est-ce fini ? Non ! Chaque fonction appelée se terminant retourne sa valeur au programme ou sous-programme l'ayant appelé. Donc cela continue :

- 6ème étape : fact(2) : 2*fact(1)=2, retourne 2 à fact(3)
- 7ème étape : fact(3) : 3*fact(2)=6, retourne 6 à fact(4)
- 8ème étape : fact(4) : 4*fact(3)=24, retourne 24 à fact(5)
- 9ème étape : fact(5) : 5*fact(4)=120, retourne 120 au programme appelant.

Si vous suivez l'ordre des appels<->retours vous obtenez le schéma suivant :



Les allers et retours d'un appel récursif

Un algorithme un peu plus correct de fact() est donc :

```
Fonction fact(n :entier) :entier
Début
  Si n>1 Alors
    n←n*fact(n-1)
  FinSi
  Retourne n
Fin
```

Il manque encore le cas de la factorielle de zéro, qui vaut un. Au final, la fonction récursive fact() pourrait ressembler à ceci, car une factorielle n'est possible qu'avec des entiers positifs :

```
Font fact(n:entier):entier
Début
  Si n=0 Alors
    Retourne 1
  Sinon
    Retourne n*fact(n-1)
  FinSi
Fin
```

La même fonction en PHP avec son programme d'accompagnement :

```
<html>
  <head><meta/>
    <title>Factorielle</title>
  </head>
  <body>
  <?php
  function fact($n) {
    if($n==0) return 1;
    else return $n*fact($n-1);
  }

  $n=fact(10);
  echo "$n";
?>
  </body>
</html>
```

3. Un exemple pratique : les tours de Hanoi

Les tours de Hanoi sont un jeu de réflexion qui a été inventé en 1883 par N. Claus de Siam professeur au collège de Li-Sou-Stian. Si la curiosité vous a poussé à rechercher ces noms et villes, peut-être avez-vous eu une surprise : aucun des deux n'existe. Ce sont en fait des anagrammes faisant croire que ce jeu a été inventé par un asiatique. Tout faux ! N. Claus de Siam est l'anagramme de Lucas D'Amiens, (Edouard Lucas en fait), né à Amiens et Li-Sou-Stian est l'anagramme de Saint-Louis, nom du Lycée où Lucas enseignait.

Les tours de Hanoi dérivent d'une légende Hindou qui dit qu'un temple dispose de trois poteaux sur lesquels s'empilent 64 disques en or de diamètres différents. Les prêtres de Brahma déplacent continuellement les disques du premier poteau vers le troisième en passant éventuellement par un poteau intermédiaire et en respectant quelques règles simples :

- Ils ne peuvent déplacer qu'un seul disque à la fois.
- Ils ne peuvent déplacer un disque que dans un emplacement vide ou sur un disque de plus grand diamètre.

La légende dit aussi que les disques ont été empilés au début des temps et que lorsque les prêtres auront fini de les déplacer, ce sera la fin du monde.

Nul doute que si les prêtres de la légende avaient eu un ordinateur, nous serions tous déjà morts… Quoique ! Avec 64 disques il faut 2^{64}-1 déplacements (18446744073709551615 et uniquement sans se tromper), soit à raison de un par seconde 584 542 046 090 ans (584 milliards d'années). Sachant que l'univers a environ 14 milliards d'années (c'est théorique), il nous reste 570 milliards d'années pour en profiter. En plus, les disques en or doivent être très lourds à déplacer.

Le jeu fait référence à Hanoï car dans la capitale du Vietnam, une ancienne colonie française, les toits de certaines pagodes ont la forme de plateaux empilés.

L'algorithme récursif pour résoudre ce problème est un grand classique nécessitant un peu de torture mentale. Supposons que vous sachiez déplacer n-1 disques. Pour en déplacer n, il suffit de déplacer (n-1) disques du piquet 1 au piquet 3, puis de déplacer le grand disque du piquet 1 au piquet 2 et de terminer en déplaçant les (n-1) autres disques du piquet 3 vers le piquet 2.

Soient :

- n le nombre de disques,
- a la position de départ, valant 1,
- b, la position d’arrivée, valant 2,
- c, la position intermédiaire utilisable, valant 3,
- au départ, tous les disques sont à la position a,
- une procédure "déplacer" qui déplace un disque d’une position à une autre via une position intermédiaire.

L’algorithme est le suivant :

```
Procédure déplacer(n,a,b,c :entiers)
Début
  Si n>0 Alors
  déplacer(n-1,a,c,b)
  Afficher "De ",a, "vers ",b
  déplacer(n-1,c,b,a);
FinSi
Fin
```

Soit en PHP :

```
<html>
  <head><meta/>
    <title>Tour de Hanoi</title>
  </head>
  <body>
  <?php
  function deplace($n, $a, $b, $c)
  {
    if($n>0) {
      echo "deplace(".($n-1).",$a,$b,$c)<br />";
      deplace($n-1,$a,$c,$b);
      echo "De $a vers $b<br />";
      echo "deplace(".($n-1).",$c,$b,$a)<br />";
      deplace($n-1,$c,$b,$a);
    }
  }
  deplace(3,1,2,3);

?>
  </body>
</html>
```

Les différents fichiers

1. Préambule

Vous permettrez, pour une fois, d'être un peu direct : les fichiers sont une calamité à décrire en algorithmique. Durant des années, jusqu'au milieu des années 1990, le COBOL a régné en maître chez les langages de programmation en informatique de gestion. Ce langage était aussi puissant dans ce domaine que sa syntaxe était pratique. Ceux d'entre vous qui connaissent un peu le COBOL savent de quoi il est question ici. Une des sources de sa puissance était sa capacité à gérer les fichiers et les enregistrements structurés, dans tous les sens, dans toutes les formes.

Seulement, le COBOL est passé de mode pour les nouveaux développements. Les multitudes de fichiers de données diverses et variées ont été remplacées par d'autres structures, notamment par des bases de données relationnelles, rendant caduque l'étude de la plupart des types de fichiers au profit de langages comme le SQL. Pourtant, les bases de données elles-mêmes sont souvent stockées dans des fichiers.

Aussi, ce chapitre est plus court que les autres. Il serait possible de décrire une foule de types de fichiers : des livres complets existent sur ce thème, de plusieurs centaines de pages. Les bases et définitions théoriques seront à peu près toutes couvertes, mais seuls seront traités les fichiers textes. PHP ne sachant pas directement (vous pouvez bien entendu tout reprogrammer, c'est son rôle) gérer des fichiers de type indexés, cela résout singulièrement le problème.

2. Problématique

Qu'est-ce qu'un fichier ? C'est exactement la même chose que dans la réalité. Quand vous avez une seule information à retenir dans votre tête, c'est facile. Quand vous en avez des milliers, c'est une mission impossible. Vous allez utiliser un agenda, rempli de fiches. Le cerveau a certes des capacités incroyables, mais comment retenir toutes les transactions bancaires de millions de comptes pour une banque ?

À cela, ajoutez deux problèmes :

- Malgré la puissance de vos ordinateurs, la capacité mémoire reste souvent limitée. Un ordinateur 32 bits "standard" (bureau, jeu) est souvent limité à 4 Go. Parfois des machines vont beaucoup plus haut, jusqu'à 64 Go. Or, même au modeste niveau de l'ordinateur personnel cela reste souvent très insuffisant : il est impossible de conserver toutes les données en mémoire. Par exemple, un film au format DV d'un camescope numérique occupe près de 20 Go, un DVD complet plus de 8 Go, etc.

- Même s'il était possible de conserver toutes les données en mémoire centrale, ce qui serait somme toute l'idéal vu la vitesse d'accès aux informations, cette dernière est volatile. C'est-à-dire que tant que l'ordinateur est allumé, que vos programmes ne plantent pas et que l'électricité fonctionne, il n'y a aucun problème. Dès qu'on débranche, tout disparaît.

Il faut donc trouver une solution pérenne pour conserver vos données. Un fichier correspond à l'une des possibilités.

3. Définition

Un fichier est un ensemble d'informations, généralement structurées d'une manière ou d'une autre. Certes la mémoire contient aussi ce genre d'informations (vous avez vu les types structurés), mais on parle de fichiers quand ces informations sont placées sur un support moins volatile que la mémoire de l'ordinateur. Aussi un fichier se place sur une disquette (en voie de disparition), une bande magnétique (comme les sauvegardes sur bande par exemple), une clé USB et surtout, sur un disque dur.

4. Les formats

a. Types de contenus

Un fichier se distingue des autres par quelques attributs dont son nom et sa catégorie. Ils se distinguent aussi entre eux par l'organisation de leurs données ce qui définit leur format. Vous connaissez probablement plusieurs formats de fichiers :

- les fichiers texte ;

- les documents des traitements de texte ;
- les fichiers son MP3 ;
- les vidéos DivX ;
- les fichiers HTML ;
- les images JPEG ;
- les fichiers de votre gestion de compte bancaire ;
- etc.

Comment s'y retrouver ? Il existe autant d'organisations de données qu'il existe de logiciels ! C'est d'ailleurs souvent un très gros problème, c'est ce qu'on appelle un format de fichiers propriétaire : il est souvent impossible de relire un format donné avec un produit concurrent.

Cela n'a pas forcément de rapport direct, mais un peu de lobbying dans ce cas précis n'est pas du superflu : il existe de nombreux formats de fichiers dit "ouverts", dont l'organisation des données qui y sont contenues est connue et documentée. Tout format de fichier "ouvert" est reconnu généralement de la même manière par les programmes sachant les utiliser. Le PDF (pour certains documents) est un format ouvert, inventé par Adobe, le format OGG Vorbis en est un autre, concurrent du MP3. Il existe encore une différenciation entre les formats propriétaires et les formats libres. Tous les formats libres sont ouverts, mais seulement très peu de propriétaires le sont. Un format libre, ouvert et répandu est un gage de pérennité et de compatibilité de vos données. OGG Vorbis est libre, PDF ne l'est pas. Si demain Adobe en développe une nouvelle version, elle pourrait être incompatible avec les précédentes et payante. On a vu des documents Word ne plus pouvoir être ouverts d'une version à une autre…

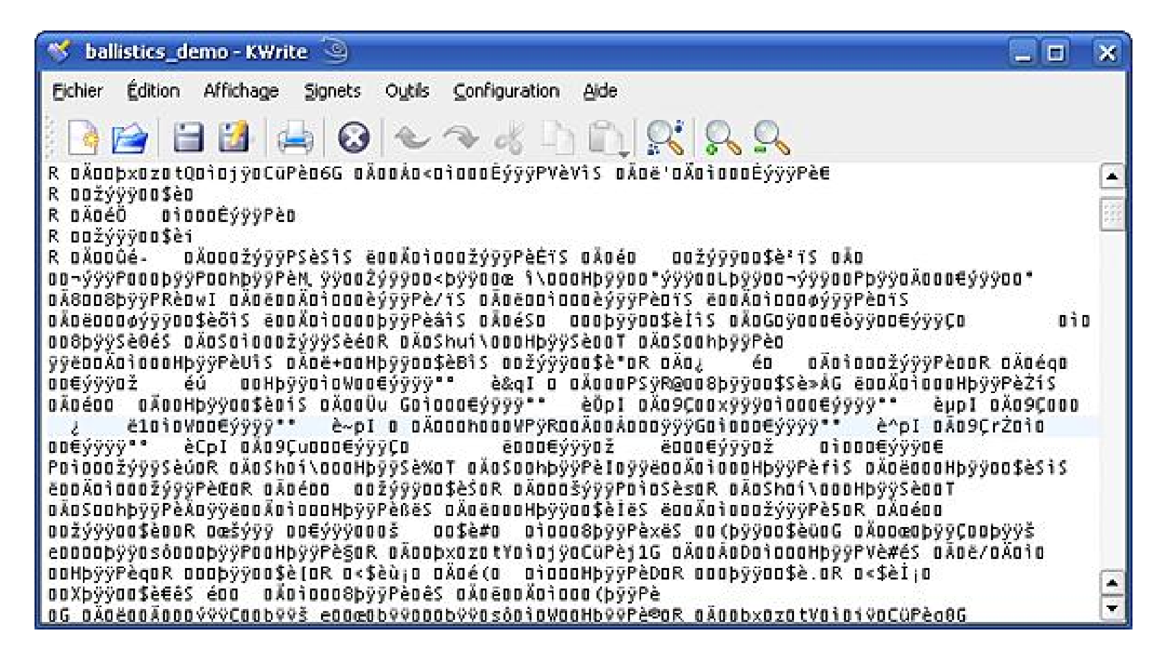
Tout ce qui peut être formalisé peut être stocké dans un fichier. Il y a cependant des différences évidentes et flagrantes entre un fichier qui doit stocker une photo et un autre qui stocke une page HTML d'un site web.

Deux catégories de fichiers sont distinguables :

- Les fichiers organisés sous forme de lignes de texte successives, qui s'appellent des fichiers texte.
- Les fichiers bruts, contenant des données variées dont des nombres représentés sous forme binaire, ne correspondant peu ou pas du tout à du texte, qui s'appellent des fichiers binaires.

b. Le fichier binaire

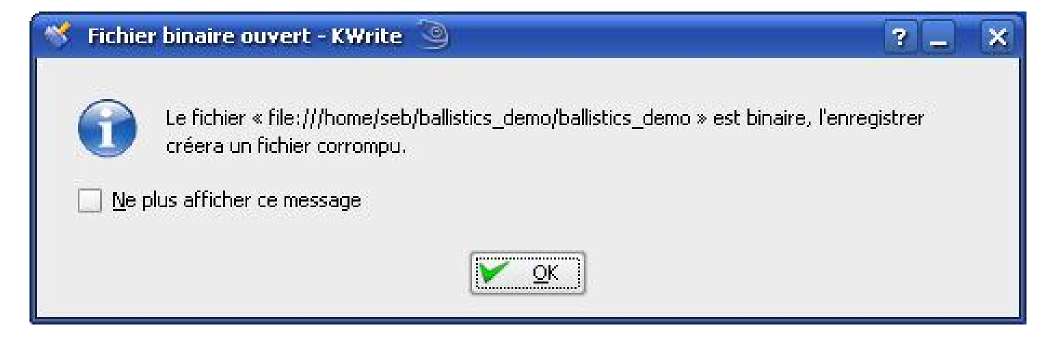
Comment reconnaître ces fichiers les uns des autres ? C'est très facile ! Prenez un éditeur de texte simple (notepad sous Windows) et ouvrez le fichier sur lequel vous avez des doutes. Si vous obtenez ceci (l'éditeur est kWrite sous Linux) :



Résultat de l'ouverture d'un fichier binaire dans un éditeur

C'est que vous êtes en présence d'un fichier binaire, en l'occurrence ici un programme compilé. C'est illisible directement, c'est une suite d'instructions et de données compréhensibles par le microprocesseur. Pourtant, sorti sous forme hexadécimale ou binaire, il serait possible de convertir les séquences en instructions lisibles, c'est le rôle d'un désassembleur. Même principe pour un fichier au format MP3 : ce format nécessite un programme qui va analyser le format connu et en effectuer la transformation en ondes sonores qui pourront être envoyées vers votre carte son, puis vos oreilles.

Il est intéressant de noter que l'éditeur de texte kwrite ayant été utilisé pour la capture ci-dessus a fourni un avertissement intéressant :



Risque de corruption d'un fichier binaire

Un fichier binaire n'utilise pas de représentation ASCII ou Unicode pour représenter les nombres, d'où les caractères farfelus affichés. Aussi réenregistrer un fichier binaire avec un éditeur de texte cassera le format du fichier, rendant ce dernier totalement inutilisable.

La configuration complète de Windows est placée dans un registre ou base de registre. Cette base est stockée dans deux fichiers, system.dat et users.dat, au format binaire. S'ils sont corrompus, il est fortement probable de devoir tout réinstaller. L'accès à ce registre ne vous est qu'indirectement possible via l'éditeur de base de registre regedit ou regedt32, mais pas en l'ouvrant sous notepad.

---

c. Le fichier texte

Un fichier texte est ce qu'il annonce : il contient du texte sous forme de lignes. Chaque ligne est distincte des autres. Pour savoir s'il doit passer à la ligne l'ordinateur ou plutôt le système d'exploitation ajoute des caractères spéciaux. Sous Windows, ce sont les caractères ASCII CR (*Carriage Return*, retour chariot) qui vaut 13 et LF (*Line Feed*, Saut de Ligne) qui vaut 10. Cette séquence **CRLF** passe à la ligne. Sous Unix et ses dérivés, dont MacOS, seul le caractère LF est nécessaire (une conversion est nécessaire pour convertir du texte de Windows vers Unix et réciproquement). Nul besoin ici de vous montrer une capture d'un éditeur ouvrant un fichier texte…

D'où une qualité essentielle du fichier texte :

Un fichier texte est directement lisible par l'utilisateur et son organisation devient évidente.

Dans un fichier texte, même les nombres sont représentés sous forme de caractères ASCII (ou unicode, selon le cas). Le nombre 1234 sera représenté exactement sous cette forme caractère, soit la suite ASCII 31-32-33-34 (héxadécimal), 49-50-51-52 (décimal) ou encore 0011001 0011010 0011011 0011100 en binaire. Le même nombre dans un fichier binaire serait représenté en 04D2 (héxadécimal) ou 10011010010 en binaire : une grosse différence.

Cela veut dire que quand vous allez écrire des nombres dans un fichier texte, vous devrez utiliser des fonctions de conversion de nombres vers des chaînes. Réciproquement, vous devrez utiliser une autre fonction quand vous lirez ces chaînes pour les convertir en nombres. Vous pouvez utiliser les fonctions prédéfinies pour cela :

- `x=chnum(txt)` : convertit la chaîne txt en valeur numérique
- `txt=numch(x)` : convertit la valeur numérique x en chaîne

Vous avez pu voir dans quelques exemples des chapitres précédents que PHP dispose d'un arsenal intéressant de fonctions pour convertir du texte en entier, réel, etc. Un souci peut se poser avec les réels : les anglo-saxons (et d'autres) utilisent le point comme séparateur décimal, alors que vous utilisez probablement la virgule en France. Certes les langages de programmation utilisent principalement le point, mais les tableurs francisés utilisent la virgule. Amis canadiens, si vous lisez ce livre, vous êtes plus chanceux que les français, qui devront faire attention en convertissant une chaîne en nombre et vice versa, à cause de la virgule. Dans le doute, lisez le manuel de votre langage.

Quasiment tous les fichiers de configuration d'un système d'exploitation Unix sont des fichiers texte. Pour modifier cette configuration, il suffit donc bien souvent de l'ouvrir avec un éditeur de texte, de modifier ou d'ajouter des lignes, de sauver et c'est tout.

---

d. Quel format utiliser ?

Faites preuve de bon sens. Il serait ridicule d'utiliser un fichier texte pour sauver une image, tout comme il serait ridicule d'utiliser un fichier binaire uniquement pour sauver du texte simple (un format de traitement de texte est plus complexe qu'il n'y paraît). Des formats réputés (à la mode) comme le XML sont des fichiers à la structure pouvant devenir très complexe et pourtant ce sont des fichiers texte.

Fichier texte

Les propriétés d'un fichier texte sont les suivantes :

- Les fichiers texte sont utilisés pour stocker des données structurées.
- Ces données peuvent être du texte, des nombres, du moment que tout est converti en texte lisible.
- Ces données structurées sont des enregistrements.
- Les enregistrements d'un fichier texte sont représentés sous forme de lignes, séparées les unes des autres par une séquence CRLF (Windows) ou LF (Unix/Mac).
- Chaque ligne représente une structure d'enregistrement, selon un format préétabli (fixe ou délimité).
- Un fichier texte est lisible et modifiable par n'importe quel éditeur de texte.
- L'interprétation des enregistrements dépend bien entendu de sa finalité.
- Un fichier texte ne peut être lu que ligne à ligne.
- Un enregistrement s'ajoute uniquement à la fin du fichier. Pour modifier ou insérer un enregistrement, il faudra probablement tout réécrire.

Fichier binaire

Comme pour les fichiers texte, voici quelques propriétés d'un fichier binaire :

- Un fichier binaire peut stocker n'importe quoi, que ce soit structuré ou non.

- Les fichiers binaires n'ont pas de structure apparente, ils représentent une suite de données (octets) écrites les unes après les autres.

- Toutes les données sont représentées sous forme binaire. Les nombres sont convertis et s'il y a du texte, il apparaît comme tel mais non structuré. De même si on y voit un caractère, il se peut que ce soit un effet de bord : la conversion d'un nombre donnant "par hasard" quelque chose de lisible via les codes ASCII ou Unicode mais n'ayant aucun rapport. Pour résumer : les données d'un fichier binaire y sont écrites exactement comme si elles sortaient de la mémoire : c'est la même représentation.

- La structure des enregistrements est dépendante de l'interprétation du programme. Les enregistrements peuvent être de longueur fixe, mais collés les uns après les autres sans retour à la ligne.

- De ce fait, un fichier binaire ne doit pas être ouvert ou enregistré depuis un éditeur de texte : il est souvent illisible. Seul le programme sachant le manipuler est apte à l'utiliser. Vous pouvez cependant utiliser un éditeur héxadécimal.

- Le fichier peut être lu octet par octet, ou par bloc, ou entièrement, depuis n'importe quelle position, puisque c'est vous qui définissez sa structure. Idem pour les enregistrements.

5. Les accès aux fichiers

a. Séquentiel

Le fichier séquentiel permet d'accéder aux données dans leur ordre d'écriture. Vous accédez aux données les unes après les autres. Pour pouvoir accéder au millième enregistrement, vous devez d'abord lire les 999 premiers (ce qui ne veut pas dire que vous devez les interpréter). Les fichiers texte sont généralement des fichiers séquentiels, chaque enregistrement étant représenté par une ligne.

Un fichier binaire peut très bien être séquentiel, puisque encore une fois c'est vous qui déterminez sa structure. Vous pouvez décréter que les n premiers octets sont la description d'une figure géométrique de n faces, puis que les n autres enregistrements représentent la longueur des faces, les angles, etc.

Il n'est pas possible de modifier directement l'enregistrement d'un fichier séquentiel. Vous pouvez ajouter un enregistrement à la fin. Pour supprimer, vous pourrez utiliser un éditeur.

b. Accès direct

L'accès direct est aussi appelé aléatoire. Il n'y a rien de spécialement aléatoire de votre côté, mais contrairement à l'accès séquentiel, vous pouvez sauter directement à l'endroit que vous désirez. Pour un fichier texte, cela pourrait être le numéro d'enregistrement (de ligne). Pour un fichier binaire, c'est la position de l'octet souhaité.

c. Indexé

Dans un fichier indexé, les enregistrements sont identifiés par un index qui peut être un numéro ou une valeur quelconque, un identifiant. La connaissance de cet identifiant permet d'accéder directement à l'enregistrement qu'il référence. Les enregistrements sont souvent placés les uns à la suite des autres dans le fichier, comme en séquentiel. Les index sont eux placés dans un tableau d'index, avec pour chaque index la position de l'enregistrement correspondant dans le fichier. Un fichier indexé est donc un super mélange des deux précédents.

Les index peuvent être totalement indépendants, auquel cas il n'y a pas forcément de moyen de lire les enregistrements sans connaître son index. Mais bien souvent, vous entendrez parler de séquentiel indexé : le fichier est indexé, mais depuis un index, vous pouvez lire successivement tous les enregistrements suivants. Les index peuvent être chaînés dans un sens, dans les deux, voire triés, etc. La notion de chaînage est l'un des sujets du prochain chapitre.

d. Autre ?

Un langage comme le COBOL est le roi de la manipulation des fichiers, notamment en séquentiel indexé. Il serait plus complexe, mais pas infaisable de décoder un fichier MP3 dans ce langage.

Vous pouvez faire ce que vous voulez d'un fichier, c'est vous qui en déterminez la structure. Mieux (ou pire, selon le

cas) : le langage de programmation n'a que faire du contenu du fichier. Si vous demandez au langage C d'ouvrir en mode texte un fichier binaire et que vous en lisez successivement les enregistrements, il ne bronchera pas : sur un gros fichier, il trouvera probablement quelques retours à la ligne.

De même, vous pouvez sauter à l'octet de votre choix dans un fichier texte, en lire pile trois octets et c'est tout. C'est à vous d'implémenter l'interprétation du contenu des fichiers dans le langage visé. Quant à l'indexé, c'est une catastrophe, PHP, C, C++ et Java ne proposent rien de prédéfini. C'est à vous de tout créer avec les fonctions de bases existantes.

Les enregistrements

Ce chapitre se concentre sur les fichiers texte. Dans ceux-ci, vous devez définir la structure de vos enregistrements. Bien que tout soit possible, notamment avec des structures en arborescence comme le XML, vous pouvez initialement choisir entre deux méthodes assez simples : les enregistrements avec délimiteurs ou à largeur fixe.

1. Les délimiteurs

Les enregistrements délimités sont courants sous Unix. Dans chaque ligne, les valeurs individuelles sont appelées des **champs** et sont séparées entre elles par un caractère particulier appelé caractère de séparation ou caractère de délimitation, ou enfin délimiteur. N'importe quel caractère peut convenir, cependant il ne doit pas se retrouver dans la valeur d'un champ, ce qui aurait pour effet de casser la structure de l'enregistrement. Ce sont souvent le point-virgule ";" ou les deux points ":" qui sont utilisés.

Les champs d'enregistrements sont généralement encadrés par des guillemets lorsqu'il s'agit de chaînes de caractères, rien pour des valeurs numériques. Ce n'est pas une affirmation : ce n'est pas toujours le cas et c'est à vous de gérer le type d'une valeur donnée, au sein de votre programme.

Voici un simple exemple de contenu de fichier délimité de type courant sous Unix :

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:Daemon:/sbin:/bin/bash
lp:x:4:7:Printing daemon:/var/spool/lpd:/bin/bash
mail:x:8:12:Mailer daemon:/var/spool/clientmqueue:/bin/false
news:x:9:13:News system:/etc/news:/bin/bash
uucp:x:10:14:Unix-to-Unix CoPy system:/etc/uucp:/bin/bash
```

Les plus perspicaces d'entre vous auront reconnu un morceau du fichier /etc/passwd qui contient quelques informations sur les comptes des utilisateurs du système Unix. Chaque ligne est un enregistrement dont la structure est la suivante :

- Un séparateur ":" délimite les différents champs.
- Il y a sept champs, numérotés (par convention) de 1 à 7.
- 1er champ : nom de l'utilisateur (son login).
- 2ème champ : indicateur de mot de passe (ici stocké ailleurs).
- 3ème champ : UID, identifiant numérique unique de l'utilisateur.
- 4ème champ : GID, identifiant du groupe de l'utilisateur (stocké ailleurs).
- 5ème champ : commentaire libre.
- 6ème champ : dossier personnel de l'utilisateur.
- 7ème champ : l'interpréteur de commandes (shell) de connexion.

Vous voyez bien qu'il est possible de placer des informations très importantes dans un fichier texte.

La manipulation d'un tel fichier est assez évidente : il suffit de lire une ligne, puis de découper celle-ci champ par champ, ce qui est plutôt simple car il suffit dès lors de trouver les délimiteurs. La plupart des langages proposent des fonctions qui permettent de découper une chaîne selon des délimiteurs. Ce type de fichier a aussi bien des avantages que des inconvénients :

- Du fait que les champs soient collés les uns aux autres et que chaque champ ne prend pas plus d'espace que sa donnée occupe, fournit un réel avantage en terme d'occupation d'espace disque et de mémoire.
- Cependant, le traitement de découpage de la chaîne en fonction de la position forcément aléatoire d'un

délimiteur est plus complexe qu'il n'y paraît. Certes les langages peuvent fournir des fonctions appropriées, mais comment feriez-vous vous-même ?

Sur ce dernier point c'est assez facile à formaliser sous forme d'algorithme. Une fonction "split" reçoit trois paramètres : une chaîne de caractères, le délimiteur et une position de départ. Elle retourne le champ à partir de cette position, qui est le numéro de caractère de la chaîne, démarrant à un. Si la fonction retourne une chaîne vide, c'est qu'il n'y a plus rien.

```
Fonction split(txt :chaîne,delim :caractère,pos :entier) :chaîne
Var
  l,i :entiers
  tmp :chaîne ;
Début
  l=longueur(delim)
  tmp←""
  i←pos
  Tant que i<=l et milieu(txt,i,1)!=delim Faire
    tmp←tmp&milieu(txt,i,1) // un seul caractère concaténé
    i←i+1
  FinTantQue
  Retourne tmp
FinFonct
```

Pour exploiter cette fonction, voici un petit programme qui recherche successivement tous les champs d'une ligne. Il suffit de boucler tant que la fonction split ne retourne pas de chaîne vide. À chaque passage dans la boucle, il faut incrémenter la position de la longueur de la chaîne trouvée, plus 1 (le délimiteur), pour se trouver sur l'éventuel nouveau champ…

```
Programme decoupe
Var
  ligne,result :chaîne
  pos :entier
Début
  pos←1
  ligne←"root:x:0:0:root:/root:/bin/bash"
  result←"toto"
  Tant que result!="" Faire
    result←decoupe(ligne," :",pos)
    Si result!="" Alors
      Afficher result
      pos←pos+longueur(result)+1 //après le délimiteur
    FinSi
  FinTantQue
Fin
```

Vous constatez que le traitement n'est pas anodin et sur des milliers d'enregistrements cela peut compter.

2. Largeur fixe

Dans des enregistrements à largeur fixe, il n'y a pas de délimiteurs. Chaque champ a une longueur prédéfinie et occupe toute cette longueur, quitte à être complété par des espaces. Les champs sont ainsi collés les uns aux autres, en un seul gros bloc.

S'il fallait convertir une ligne ci-dessus en enregistrement fixe, il faudrait savoir que :

- Un login fait en principe 8 caractères.
- Un mot de passe 1 seul (stocké ailleurs).
- Les UID et GID utilisent au maximum 5 chiffres.
- Le commentaire sera arbitrairement (pour l'exemple) de 15 caractères.
- Les chemins et shells arbitrairement aussi à 15 caractères.

```
|--------------------------------------------------------------------|
|root     x0     0    Commentaire     /root          /bin/bash       |
|bin      x1     1    Commentaire     /bin           /bin/bash       |
|daemon   x2     4    Commentaire     /sbin          /bin/bash       |
|rohaut   x123   123  Compte Seb      /home/seb      /bin/ksh        |
|--------------------------------------------------------------------|
|1      891    11    22               33             45             6|
|         0    45    01               56             90             4|
```

Deux choses sautent aux yeux :

- Contrairement au format délimité, le format à largeur fixe consomme bien plus de mémoire. Dans cet exemple 64 octets sont utilisés pour chaque enregistrement. Dans le format délimité, la taille est variable, mais la première ligne n'utilisait que 32 octets.

- Cependant, la récupération de tels enregistrements est bien plus simple, car vous connaissez à l'avance la taille de chaque champ et donc toutes les positions pour découper vos enregistrements. Par contre, il faudra penser à supprimer les éventuels espaces en trop, en supprimant les espaces finaux. Les langages proposent souvent une fonction appelée "trim" qui le fait pour vous. Nul besoin donc de faire des recherches comme précédemment.

Cette praticabilité compense l'utilisation de la mémoire, d'autant plus qu'elle coïncide fortement avec les types structurés abordés dans le chapitre Les tableaux et structures. La lecture et l'écriture dans les fichiers se trouvent ainsi facilitées.

Un fichier à format fixe a aussi un autre avantage : il est possible de choisir parmi les deux formats, texte ou binaire, pour l'enregistrer. Autrement dit soit ligne à ligne, soit contigus. Comme vous connaissez la taille exacte d'un enregistrement, il suffit de dire que les octets 1 à 64 représentent le premier, 65 à 129 le deuxième, 130 à 194 le troisième et ainsi de suite. Vous devrez cependant faire attention avec les valeurs numériques, un entier occupant généralement quatre octets, un réel double précision huit, etc.

Connaissant à l'avance les positions de chaque champ, vous utiliserez les sous-programmes prédéfinis de chaque langage pour découper une sous-chaîne de caractères équivalent à la fonction milieu du chapitre précédent.

3. Principes d'accès

a. Étapes de base

Pour travailler avec des fichiers, vous devrez respecter un certain ordre. Il vous faudra :

- **Ouvrir** le fichier, c'est-à-dire indiquer à quel fichier vous voulez accéder et comment.

- **Traiter** le contenu du fichier : le lire, y écrire, bref toutes les opérations souhaitées pour accéder et manipuler son contenu.

- **Fermer** le fichier, quand tous les traitements sont terminés.

L'étape la plus importante est la première. Comment ouvrir un fichier ? Si vous aviez le droit d'ouvrir seulement un seul fichier, ce serait assez simple, mais qu'est-ce qui vous empêcherait d'en ouvrir trois ou quatre en même temps ?

b. Identificateurs de fichiers et canaux

Le programme doit pouvoir savoir dans quel fichier il travaille lorsqu'il s'agit d'y lire ou d'y écrire des données. Ceci passe par l'utilisation d'un identifiant unique pour chaque fichier ouvert.

Dans la pratique, tous les langages, ou presque, en interne, utilisent la même méthode. L'accès à un fichier passe par l'utilisation d'un canal. Dans la réalité, la vraie vie (sous-entendez par là que l'informatique n'est pas la vraie vie, ou plutôt un moyen mais pas une fin), un canal relie entre eux deux cours d'eau, ou à la mer, un lac, etc.

Dans ce canal transite, outre de l'eau, des bateaux, péniches, dans un sens et dans l'autre.

Dans le monde virtuel, un canal permet de faire transiter un flux d'information (les données) d'un programme vers un fichier, d'un fichier vers un programme, d'un programme à un autre, entre deux fichiers, entre un programme et un périphérique, etc. Par exemple, il existe un canal qui fait transiter ce que vous tapez au clavier vers le

programme qui attend une saisie, un autre canal pour transférer vers l'affichage ce que le programme doit afficher. Certains canaux fonctionnent dans les deux sens, d'autres non.

Chaque canal porte un numéro unique, certains ont des noms prédéfinis. Celui rattaché par défaut au clavier, appelé canal d'entrée standard, porte le numéro 0 (si tant est que 0 soit un nombre) et s'appelle STDIN. Celui qui fait transiter les informations à afficher vers l'écran (ou plutôt le pilote ou sous-système d'affichage), appelé canal de sortie standard porte le numéro 1 et s'appelle STDOUT. Il en existe un troisième appelé STDERR et portant le numéro 2, chargé de véhiculer les messages d'erreur.

La notion de canal est flagrante avec certains systèmes d'exploitation, surtout Unix qui en use et en abuse. Windows hérite lui-même de cette notion, exploitable simplement au travers de l'interpréteur de commandes d'origine DOS. En C, il est possible d'utiliser les fonctions de lecture et d'écriture de fichiers avec ces trois canaux.

Vous disposez de tous les canaux au-delà du numéro deux, donc 3 et suivants pour vos propres fichiers. L'ouverture d'un fichier consiste donc, directement ou indirectement, à lui rattacher un canal. Les données que vous écrirez dans le fichier iront de votre programme vers le fichier par ce canal et celles que vous lirez du fichier au programme toujours par ce canal.

Les notations algorithmiques peuvent varier sur ce point. Certaines notations reprennent la syntaxe d'un langage de type Basic ou Visual Basic, dans lequel vous devez indiquer vous-même quel numéro de canal utiliser à l'ouverture du fichier. D'autres reprennent une syntaxe issue du C où c'est la fonction C ouvrant le fichier qui choisit un canal et vous récupérez un identifiant de fichier sous forme de variable. Cet identifiant est le **nom logique** du fichier au sein du programme. C'est cette notation qui est généralement préférable en algorithmique.

En C, une fonction d'ouverture de fichier retourne une variable de type FILE qui est en fait un type structuré contenant diverses informations dont un entier contenant un numéro qui se révèle être le numéro du descripteur de fichiers au niveau du système d'exploitation. Des limites sont définies au sein du système. Sous Linux, il ne peut y avoir par défaut plus de 1024*1024 fichiers ouverts, mais en pratique 1024 par processus (programme). Cela devrait suffire, cependant divers mécanismes permettent d'aller encore plus loin, généralement jusqu'à 8192. Au-delà, cela devient très très lourd. Un nom logique de fichier est donc un enregistrement de type structuré.

---

c. Les modes d'ouverture

En ouvrant un fichier, il faut indiquer comment vous souhaitez y accéder. Souhaitez-vous seulement lire son contenu, y écrire où vous voulez, ou ajouter des lignes à la fin ?

- En **lecture**, vous avez un accès en lecture seule au fichier. Vous ne pouvez pas y écrire. Vous pouvez vous y déplacer, retourner au début, lire tout ce que vous voulez, mais c'est tout.
- En **écriture**, parfois nommé lecture/écriture, vous pouvez modifier n'importe quelle partie du fichier, écrire où vous voulez, écraser vos anciennes données qui seront donc définitivement perdues. Attention à la casse si vos enregistrements sont de taille variable dans un fichier texte ! Vous pouvez aussi lire le contenu du fichier.
- En **ajout** (append), vous ne pouvez pas lire le fichier, mais uniquement ajouter des données après la fin de celui-ci, comme ajouter un enregistrement ou quelques octets tout au bout. Vous ne pouvez pas modifier les données qui y sont déjà écrites.

Le problème principal que vous rencontrerez quand vous utiliserez des enregistrements dans un fichier, c'est qu'il n'y a aucun moyen d'effacer un enregistrement, donc de réduire la taille de celui-ci. Vous devrez donc ruser, utiliser un indicateur particulier pour indiquer un enregistrement supprimé et prévoir des traitements de réorganisation des fichiers (réindexation, suppression des lignes, listes chaînées, etc) en passant probablement par des fichiers temporaires, ou tout charger en mémoire, puis réécrire seulement les bonnes valeurs…

Fichier texte séquentiel

1. Ouvrir et fermer un fichier

Vous devez tout d'abord déclarer le nom logique du fichier, la variable qui permettra, associée au fichier, de travailler avec. Ceci se place comme d'habitude dans la section Var. Vous indiquez ensuite le type d'accès : séquentiel, direct, indexé, etc. Notez qu'en accès direct, vous pouvez aussi travailler en mode séquentiel.

```
VAR
  fic:fichier séquentiel
```

Vous devez ensuite ouvrir le fichier, en associant nom logique et nom du fichier, avec le mode d'ouverture souhaité. Ceci s'effectue avec l'instruction **Ouvrir**.

```
Ouvrir "toto.txt" dans fic en lecture
```

Vous trouverez aussi parfois cette syntaxe sous forme de fonction, plus proche de certains langages :

```
fic←Ouvrir("toto.txt","lecture")
```

La première syntaxe est souvent préférable en algorithmique. Pour fermer un fichier, utilisez l'instruction **Fermer** suivie du nom logique du fichier.

```
Fermer fic
```

Il existe aussi une syntaxe fonctionnelle, mais les mêmes remarques s'appliquent : utilisez la première en cas de doute.

```
Fermer(fic)
```



Ne placez pas le nom du fichier dans les instructions de lecture, écriture et de fermeture. Le programme ne connaît pas le nom du fichier, mais seulement son nom logique.

Aussi le début d'un éventuel programme ressemblerait à ceci :

```
Programme OUVRE
Var
  fic :fichier séquentiel
  nom :chaîne
Début
  nom←"toto.txt"
  Ouvrir nom dans fic en lecture
  /* traitements */
  Fermer fic
Fin
```

2. Lire et écrire des enregistrements

a. Lecture

Pour simple rappel, les enregistrements sont ici les lignes d'un fichier texte, un enregistrement étant équivalent à une ligne. La lecture d'une ligne se fait via l'instruction **Lire**. Lire lit l'enregistrement présent à la position actuelle du fichier, puis se place sur l'enregistrement suivant. À l'ouverture du fichier, Lire lit la première ligne. Un nouveau Lire lira la deuxième et ainsi de suite jusqu'à la fin. C'est pour cela que la lecture est dite séquentielle. Vous trouverez aussi la même instruction sous le nom **LireFichier**, c'est la même chose.

La syntaxe est la suivante :

```
Lire(nom_logique,variable)
```

La variable en second paramètre reçoit l'enregistrement, la ligne lue. Dans l'exemple suivant, un enregistrement du fichier des mots de passe de l'exemple ci-dessus est lu, partant du principe que la largeur est fixe, puis l'enregistrement est découpé pour récupérer le login et l'uid, converti en entier.

```
Programme OUVRE
Var
  fic :fichier séquentiel
  nom :chaîne
  ligne,login :chaîne
  uid :entier
Début
  nom←"passwd"
  Ouvrir nom dans fic en lecture
  /* traitements */
  Lire(fic,ligne)
  login←trim(milieu(ligne,1,8))
  uid←chnum(milieu(ligne,10,5))
  Afficher login,uid
  Fermer fic
Fin
```

Que faire cependant quand on ne connaît pas à l'avance le nombre de lignes du fichier ? Comment savoir si la fin du fichier a été atteinte ?

Vous avez deux possibilités :

- Selon certains formalismes algorithmiques, **Lire** est une fonction qui retourne un booléen, donc vrai ou faux. Si vous tentez de lire un enregistrement et qu'il n'y en a plus, Lire retourne FAUX. Autrement dit, tant que Lire est VRAI, on peut continuer à lire les lignes suivantes.

- La fonction **EOF()** parfois appelée aussi **FinFichier()** retourne un booléen qui indique si la fin du fichier a été atteinte ou non. Cette fonction prend en paramètre le nom logique du fichier. Elle retourne VRAI si la fin du fichier a été atteinte, donc s'il ne reste plus d'enregistrements à lire.

La fonction algorithmique **EOF()** va retourner VRAI si le fichier que vous ouvrez ne contient pas d'enregistrements, donc si la fin de fichier est atteinte dès l'ouverture. Or dans quelques langages, le programme ne le sait pas tant qu'il n'a pas tenté de lire un enregistrement et donc une fonction de ce type retournerait FAUX tant qu'aucune lecture n'aurait eu lieu. Prudence...

Le programme suivant va lire tout le fichier des mots de passe et placer les logins et UID dans des tableaux. Comme on ne connaît pas à l'avance le nombre d'éléments, on se limitera arbitrairement à 100 lignes.

```
Programme LIREFIC
Var
  fic :fichier séquentiel
  nom :chaîne
  ligne:chaîne
  i:entier
  login :tableau[1..100] de chaînes
  uid :tableau[1..100] d'entiers
Début
  nom←"passwd"
  i←0
  Ouvrir nom dans fic en lecture
  Tant que NON EOF(fic) Faire
    Lire(fic,ligne)
    i←i+1
    login[i] ←trim(milieu(ligne,1,8))
    uid[i] ←chnum(milieu(ligne,10,5))
  FinTantQue
  Afficher i," enregistrements lus"
  Fermer fic
Fin
```

b. Écriture

Lire est une chose, écrire dans un fichier en est une autre. L'écriture utilise l'instruction **Ecrire**, une fonction qui prend comme paramètre le nom logique du fichier et l'enregistrement (la ligne) à écrire. Vous trouverez tout comme l'instruction de lecture, une instruction **EcrireFichier** strictement identique :

```
Ecrire(nom_logique,enregistrement)
```

Comme vous devez écrire des enregistrements à largeur fixe, c'est à vous de vérifier que les enregistrements sont à la bonne longueur. S'ils sont trop longs, vous avez mal dimensionné vos enregistrements à l'origine (un cas régulier est une adresse à rallonge ou un nom de ville composé). S'ils sont trop courts, vous devrez leur ajouter des espaces en fin de chaîne.

Par exemple, toujours dans l'exemple des logins, que faire si celui-ci est trop court, par exemple "toto", donc quatre caractères, alors qu'il devrait en faire huit ? L'astuce consiste à rallonger le login avec des espaces. Ce n'est pas un problème pour le relire, puisque la fonction trim() restaurera son état d'origine. Il est probable que votre langage de programmation propose une fonction qui remplit toute seule les morceaux manquants, des fois il s'agit même de l'instruction d'écriture (le fprintf du C le fait très bien).

En attendant, vous pouvez programmer vous-même cette fonction algorithmique. Elle ressemble fortement à la procédure RepeteCar(). Appelez-la Formate(). Elle prendra comme paramètre le champ de l'enregistrement et la longueur attendue. Elle retournera ce même champ mais reformaté : soit coupé à la bonne longueur, soit avec des espaces ajoutés à la fin.

```
Fonction Formate(champ :chaîne, longueur :entier) :entier
Var
  nbspaces,len :entiers
Début
  len=longueur(champ)
  Si len>longueur Alors
    champ=gauche(chaîne,longueur)
  Sinon
    nbspaces=longueur-len
    Tant que nbspaces<>0 Faire
      champ=champ&" "
      nbspaces=nbspaces-1
    FinTantQue
  FinSi
  Retourne champ
FinFonct
```

Quelques formalismes algorithmiques autorisent parfois de préciser à l'avance une longueur de chaîne à la déclaration de la variable.

```
Var
  login :chaîne de 8 caractères
```

Cela laisse entendre que vous n'avez plus à vous soucier que la chaîne soit trop courte ou trop longue. C'est vrai dans l'algorithme, mais attention dans un vrai langage ! Le langage COBOL anciennement au programme du BTS était un régal à ce niveau mais en C ou en Java, la taille n'est pas précisée.

La fonction Formate() fait bien votre affaire. Vous pouvez reconstruire votre enregistrement, puis l'écrire.

```
Programme ECRIT
Var
  fic :fichier séquentiel
  nom,ligne:chaînes
  tlogin,tuid,tpwd,tgid,tcmt,thome,tshell:chaînes
  uid,gid:entiers
Début
  nom←"passwd"
  Ouvrir nom dans fic en Ajout
  login←"toto"
  uid←500
  gid←501
  ...
  /* reconstitution */
  tlogin←Formate(tlogin,8)
  tuid←Formate(numch(uid),5)
  tgid←Formate(numch(gid),5)
  tpwd←Formate(tpwd,1)
  tcmt←Formate(tcmt,15)
```

```
  thome←Formate(thome,15)
  tshell←Formate(tshell,15)
  ligne←tlogin&tpwd&tuid&tgid&tcmt&thome&tshell
  /* Ecriture */
  Ecrire(fic,ligne)
  Fermer fic
Fin
```

L'instruction **Ecrire** ajoute l'enregistrement dans le fichier, puis se place à la suite de l'enregistrement créé. Aussi si vous exécutez une nouvelle instruction **Ecrire**, le nouvel enregistrement se placera à la suite du précédent. Attention cependant au mode d'ouverture du fichier ! En mode lecture/écriture, l'enregistrement ne sera pas ajouté à la fin du fichier (contrairement au mode ajout) mais à la position courante : vous écrasez les enregistrements existants, les uns après les autres.

Un programme assez simple consiste à recopier les enregistrements d'un fichier dans un autre. Pour ajouter un tout petit peu de piment, pourquoi ne pas dire que si le champ mot de passe contient un "d", l'enregistrement doit être détruit ? Il suffit de ne pas le recopier dans le nouveau fichier.

```
Programme COPIE
Var
  fic,fic2 :fichiers séquentiels
  nom,nom2 :chaînes
  ligne,pwd:chaînes
Début
  nom←"passwd"
  nom2←"backup"
  Ouvrir nom dans fic en lecture
  Ouvrir nom2 dans fic2 en Ajout

  Tant que NON EOF(fic) Faire
    Lire(fic,ligne)
    pwd ←trim(milieu(ligne,9,1))
    Si pwd<>"-" Alors
      Ecrire(fic2,ligne)
    FinSi
  FinTantQue
  Fermer fic
  Fermer fic2
Fin
```

Traiter sur disque ou en mémoire ?

Cette stratégie de recopie a un avantage : elle n'utilise que très peu de mémoire. Elle a cependant un très gros inconvénient, elle nécessite la présence à un moment donné de deux fichiers sur le disque dur. Dans le traitement précédent, le but était de supprimer les lignes inutiles du premier fichier. Au final ce genre de traitement est en trois étapes :

- Recopie des enregistrements de **passwd** vers **backup**.
- Suppression du fichier **passwd**.
- Renommage de **backup** en **passwd**.

Cette méthode sera à privilégier sur des fichiers très imposants, plusieurs milliers (ou millions) de lignes, si la mémoire ne doit pas être trop chargée.

Une autre méthode consiste à tout traiter en mémoire. Elle se fait en deux étapes :

- Lecture intégrale du fichier passwd et stockage des lignes dans un tableau.
- Réécriture du fichier passwd avec les bons éléments du tableau.

Certains langages font une distinction entre le mode d'écriture, généralement destructeur (le fichier est purgé - vidé - avant l'ajout de données) et un mode d'ajout étendu, où le fichier peut aussi être lu… Prudence.

Cette méthode est plus rapide et plus simple. L'accès et le traitement des enregistrements en mémoire sont plus rapides que l'accès à un fichier du disque. Une fois en mémoire les données peuvent être manipulées à volonté, sans avoir à relire les enregistrements. Elle est préférable si la capacité mémoire de votre ordinateur le permet. Les gros logiciels comme les gestionnaires de bases de données relationnels chargent souvent en cache plusieurs blocs de fichiers pour accélérer leurs traitements.

Tant que les supports de stockage non volatiles seront plus lents que la mémoire, c'est ainsi qu'il faudra procéder.

3. Les enregistrements structurés

Peut-être faudrait-il dire d'ailleurs enregistrements de types structurés, tels que vus dans le chapitre Les tableaux et structures. La méthode de lecture séquentielle rencontrée dans le point précédent a mis en lumière un petit problème. Quand vous récupérez un enregistrement, vous le récupérez en entier et c'est à vous de le découper ensuite. Pour en récupérer plusieurs, vous utilisez des tableaux, un pour chaque champ de l'enregistrement.

Or dans le chapitre Les tableaux et structures vous avez pris connaissance des enregistrements de types structurés. Ces enregistrements sont eux-mêmes décomposés en champs. Pourquoi ne pas lire et écrire directement un enregistrement de type structuré dans un fichier ?

Attention : si la notation algorithmique permet la lecture et l'écriture d'enregistrements de types structurés dans un fichier, ce n'est pas le cas de tous les langages. Si le COBOL le fait très bien, ce n'est pas le cas de tous les autres, comme le C ou Java, tout au moins tel que présenté ici. Ne serait-ce parce que dans ces langages les longueurs des chaînes sont variables et les nombres représentés sous forme binaire, vous devriez soit utiliser un fichier binaire, soit tout transformer en chaîne et justifier ces dernières à la longueur voulue.

Pour utiliser des enregistrements structurés, vous devez préciser la taille exacte de chaque champ composant le type structuré.

```
Type
Structure enrpwd
  login :chaîne de 8 caractères
  pwd :chaîne de 1 caractère
  uid :chaîne de 5 caractères
  gid :chaîne de 5 caractères
  cmt :chaîne de 15 caractères
  home :chaîne de 15 caractères
  shell :chaîne de 15 caractères
FinStruct
```

La déclaration d'un enregistrement ou d'un tableau d'enregistrements et l'affectation de valeurs aux champs sont expliquées dans la partie du chapitre Les tableaux et structures qui leur est consacrée.

Pour le reste, la lecture et l'écriture d'un enregistrement fonctionnent à l'identique des enregistrements simples composés "à la main". Les mêmes fonctions sont utilisées. La nuance importante est que tous les champs sont écrits d'un coup et lus d'un coup.

- Ecrire() va ajouter un enregistrement dans le fichier, qui sera composé de tous les champs du type structuré. Comme la taille exacte est précisée, tout le texte est déjà correctement formaté. Du travail en moins.

- Lire() lit un enregistrement complet, soit tous les champs du type structuré d'un coup. Après la lecture, chaque champ contient la bonne valeur.

Dans les deux cas, le type structuré doit correspondre **exactement** au contenu du fichier et réciproquement, sinon, gare aux mauvaises surprises. L'algorithme suivant écrit un enregistrement structuré dans un fichier, puis relit ce même fichier et y récupère l'enregistrement dans une autre variable de même type.

```
Programme FicEnreg
Var
  maligne,recup :enrpwd
  fic :fichier séquentiel
Début
  maligne.login←"toto"
  pwd←"x"
  uid←"1001"
  gid←"415"
  ...
  /* Ecriture */
```

```
  Ouvrir "passwd" dans fic en Ajout
  Ecrire(fic,maligne)
  Fermer fic
  /* Relecture */
  Ouvrir "passwd" dans fic en Lecture
  Lire(fic,recup)
  Fermer fic
  /* On Obtient la même chose */
  Afficher recup.login
  Afficher recup.uid
  ...
Fin
```

Dans le même ordre d'idée, vous pouvez parfaitement utiliser un tableau d'enregistrements pour lire tout le fichier et placer tous les enregistrements en mémoire.

```
Programme litout
Var
  lignes :tableau[1..50] de enrpwd
  fic :fichier séquentiel
  i :entier
Début
  i←1
  Ouvrir "passwd" dans fic en Lecture
  Tant que NON EOF(fic) Faire
    Lire(fic,lignes[i])
    i←i+1
  FinTantQue
  Fermer fic
Fin
```

4. Exemple en PHP

PHP sait manipuler les types de fichiers binaires ou texte, séquentiels ou en accès direct. Ce n'est pas très compliqué.

Le programme PHP suivant est un exemple de lecture et de réécriture d'enregistrements avec délimiteurs, comme le fichier des utilisateurs d'Unix. Il permet :

- Ouverture du fichier en lecture.
- Lecture de chaque ligne.
- Chaque ligne est placée dans un tableau à deux dimensions : la première est la ligne, la seconde les champs découpés via la méthode split().
- Une place est réservée pour d'éventuels traitements sur les enregistrements.
- Ouverture du fichier de sortie en écriture.
- Recomposition de toutes les lignes via une fonction recolle() prenant en paramètre un tableau de chaînes et le délimiteur.
- Écriture de la ligne reconstruite dans le fichier, avec un retour chariot.

En l'état le programme ne fait qu'une simple copie. À vous de créer les éventuels traitements entre la lecture et l'écriture. Aussi, les lignes sont placées dans un tableau de taille fixe. Peut-être serait-il plus avantageux d'utiliser des listes chaînées, comme expliqué dans le prochain chapitre.

Les instructions de déclaration, d'ouverture, lecture, écriture et fermeture sont placées en gras.

```
<?php

  function recolle($morceaux, $separateur) {
```

```
    $l=count($morceaux);
    $chaine="";

    for($i=0;$i<$l;$i++) {
      $chaine=$chaine.$morceaux[$i];
      if($i!=($l-1)) $chaine=$chaine.$separateur;
    }
    return $chaine;
  }


  $cpt=0;
  $ligne="x";

  $Fichier=fopen("/etc/passwd",'r');

  while(!feof($Fichier)) {
    // lit une ligne
    $ligne=fgets($Fichier);
    if($ligne) {
      // Split de la ligne
      $passwd[$cpt]=split(":",$ligne);
      $cpt++;
    }
  }
  fclose($Fichier);

  // eventuel traitement ici sur le tableau

  // Fichier de sortie
  $FicSort=fopen("/home/seb/macopie","w+");

  for($cpt=0;$cpt<count($passwd);$cpt++) {

  // On recolle les morceaux
    $ligne=recolle($passwd[$cpt],":");

    // On enregistre la ligne
    fwrite($FicSort,"$ligne");
  }
  fclose($FicSort);
?>
```

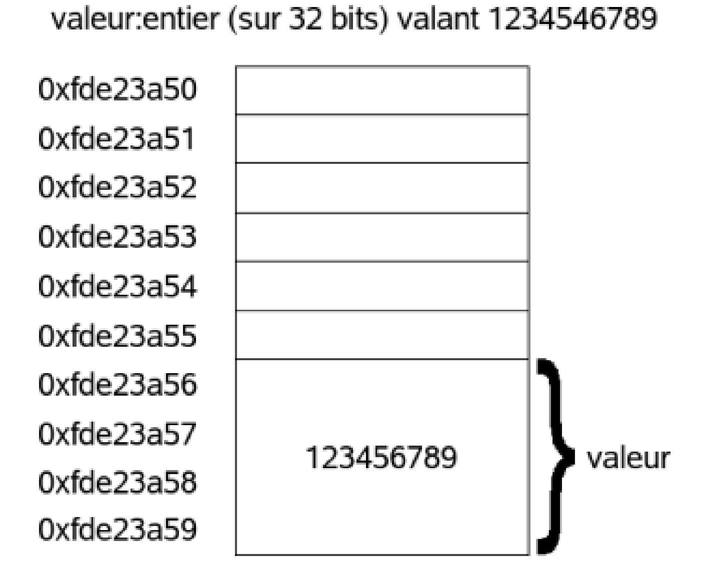
Les pointeurs et références

1. Rappels sur la mémoire et les données

a. Structure de la mémoire

Les précédents chapitres vous ont déjà appris énormément de choses sur la mémoire et l'organisation de son contenu :

- La mémoire est découpée en octets.
- Chaque octet de la mémoire dispose d'une adresse.
- Une donnée peut s'étaler sur plusieurs octets, donc occuper une plage d'adresses (par exemple 4 ou 8 octets pour un réel, plus encore pour une chaîne).



Représentation d'une variable en mémoire

Une variable est un nom donné à une ou plusieurs cases. Elle nomme la zone de la mémoire contenant la donnée. La zone de la mémoire contenant la donnée est définie par deux éléments :

- L'adresse de début de la donnée, c'est-à-dire l'adresse du premier octet contenant la donnée.
- La taille de cette zone, c'est-à-dire le nombre d'octets constituant les données.

La taille de la zone dépend du type de la donnée. Un caractère ASCII n'occupe qu'un seul octet, un entier quatre, un entier long huit, etc.

Quand vous accédez au contenu de la variable, vous accédez au contenu de la zone mémoire qui lui est associée. La variable elle-même contient donc une autre information, l'adresse de la zone mémoire à laquelle elle est attribuée.

Par convention, un ordinateur sachant gérer beaucoup de mémoire, les adresses sont notées en hexadécimal. C'est plus simple de parler d'adresse 0x2dcf0239 que de l'adresse 768541241. Certains langages permettent d'accéder et donc de voir l'adresse de la zone mémoire d'une variable, tout simplement l'adresse de la variable.

b. PHP : des limites qui n'en sont pas

Il va vous falloir légèrement déchanter. Le langage PHP utilisé depuis le début de cet ouvrage ne permet pas de connaître l'adresse de la variable. En fait, la plupart des choses décrites dans les prochaines pages seront en partie

inaccessibles à ce langage. Et parmi ces choses, la possibilité d'accéder à l'adresse mémoire des diverses variables. La raison est simple : PHP est un langage évolué de haut niveau qui n'a pas à interférer directement avec la mémoire de l'ordinateur. Le programme PHP ne sait pas où sont réellement stockées ses données en mémoire centrale, c'est l'interpréteur qui s'occupe de tout. S'il avait été possible de voir une adresse, celle-ci aurait été celle au sein de la mémoire réservée par l'interpréteur, sans pouvoir faire le lien avec la mémoire physique.

Ce système présente de nombreux avantages. Puisque c'est PHP qui gère à votre place la mémoire, vous n'avez plus à vous soucier de problèmes propres à des langages de bas niveau. En C par exemple, quand vous créez une chaîne de caractères, c'est en fait un tableau de caractères. Si vous débordez de la taille que vous avez initialement fixée, vous risquez de graves dysfonctionnements et plantages, car l'espace mémoire situé après est peut-être réservé pour une autre variable ou un bout de programme. C ne vous préviendra pas en cas d'erreur dans les indices de tableaux non plus. PHP vous débarrasse de tous ces problèmes puisqu'il gère tout cela à votre place. Aucun risque de dépasser quoi que ce soit, aucun risque d'oublier de libérer de la mémoire, etc.

Vous verrez que PHP gère très bien pour certaines choses les références, comme cela vous a brièvement été expliqué dans le chapitre "Les tableaux et structures".

c. Brefs exemples en C

Le langage C est un langage de bas niveau permettant un accès direct au contenu de la mémoire physique, tout au moins celle réservée pour votre programme. En C, vous pouvez afficher l'adresse d'une variable très simplement. Le bout de code suivant affiche le contenu et l'adresse d'une variable entière longue. Il suffit en C d'ajouter le signe "&" (ET commercial) avant le nom de la variable. La chaîne ésotérique %#xd signifie que le &i est une valeur entière (d) qui doit être convertie à l'affichage en hexadécimal, avec un préfixe 0x avant (#), le % indiquant de remplacer les caractères accolés par la variable située en paramètre, dans l'ordre.

```
long i=123456;
printf("%d à l'adresse %#xd\n",i,&i);
```

Le résultat varie évidemment d'une machine à l'autre et n'est jamais le même si on relance le programme plusieurs fois :

```
123456 à l'adresse 0xbf9f8420d
```

C permet aussi de connaître la longueur d'une variable, le nombre d'octets qu'elle utilise, avec la fonction sizeof().

```
printf("%d à l'adresse %#xd, taille de %d octets\n",i,&i,sizeof(i));
```

Une variable de type long est codée sur 4 octets sur un ordinateur 32 bits, donc le résultat ne provoque aucune surprise.

```
123456 à l'adresse 0xbfa03430d, taille de 4 octets
```

2. Le pointeur

a. Principe et définition

Que ce soit avec des langages de bas niveau, de haut niveau comme PHP ou en algorithmique, il peut être utile de trouver des moyens de manipuler directement ou indirectement des adresses de variables ou autres éléments (tableaux, types structurés, objets). En PHP ce n'est pas directement possible, vous verrez un autre moyen. Mais la notion est importante, y compris pour ce langage.

Vous n'aurez que rarement l'occasion, voire jamais, de rentrer une adresse à la main dans une variable. Même en C, vous partirez généralement d'une adresse déjà définie (celle d'une variable, d'un tableau, etc).

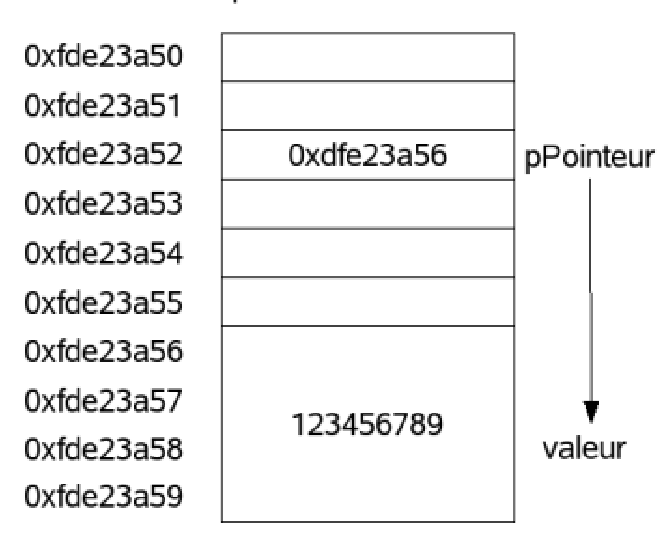
Dans les langages supportant les manipulations d'adresses mémoire, il est courant de manipuler ces adresses au travers de variables particulières qui ne contiennent non pas une donnée, mais une adresse mémoire. Ce sont des pointeurs.

Un pointeur est une variable qui contient l'adresse d'une autre variable.

Le pointeur pointe sur une autre variable dont il contient l'adresse mémoire, cette dernière étant dite variable pointée. Si vous affichez le contenu d'un pointeur, vous obtenez une adresse qui est celle de la variable pointée, tandis que si vous affichez le contenu de la variable pointée, vous obtenez la valeur associée à cette dernière.

Un pointeur est une variable. De ce fait, elle doit être déclarée, dispose elle-même de sa propre adresse en mémoire et se voit définir un type. Le type d'un pointeur ne décrit pas ce qu'il contient (c'est une adresse, donc en principe d'une longueur de 32 ou 64 bits selon les architectures) mais le type de la variable qu'il pointe. Un pointeur

sur une variable de type long devrait donc être déclaré avec un type long.



Le pointeur et la variable pointée en mémoire

b. Le C roi des pointeurs

Peut-être êtes-vous perdu dans ces définitions et explications. L'exemple suivant en C devrait suffisamment vous éclairer pour la suite. Supposez que vous voulez placer dans un pointeur l'adresse de la variable i de l'exemple précédent. Pour déclarer un pointeur en C, il suffit d'ajouter une étoile "*" avant son nom. Chaque ligne est commentée pour vous aider. L'objectif est de faire pointer un pointeur p_i sur l'adresse de l'entier i. Pour cela, le pointeur recevra cette adresse à l'aide du signe "&" devant i, car comme vu auparavant, ce signe permet d'accéder à l'adresse d'une variable.

```
long i=123456; /* Un entier long contenant 123456 */
long *p_i; /* Un pointeur sur un entier long */
/* Le pointeur p_i reçoit l'adresse de i */
p_i=&i;
printf("%d à l'adresse %#xd, taille de %d octets\n",i,&i,sizeof(i));
printf("Le pointeur p_i pointe sur l'adresse %#xd\n",p_i);
```

À l'exécution sur la machine de l'auteur, on obtient ceci :

```
123456 à l'adresse 0xbfa95cbcd, taille de 4 octets
Le pointeur p_i pointe sur l'adresse 0xbfa95cbcd
```

Notez que l'adresse de la variable i correspond maintenant exactement au contenu du pointeur p_i. Le pointeur p_i pointe sur la variable i dont il contient l'adresse.

Mais que faire de ce pointeur ? Le pointeur sert à mémoriser l'adresse, (l'emplacement mémoire) d'une autre variable. Les utilisations sont multiples et la suite du chapitre vous proposera deux applications des pointeurs : listes chaînées et arbres binaires. En attendant, depuis un pointeur vous pouvez aussi accéder au contenu de la variable pointée. Il suffit tant à l'affichage qu'à l'affectation d'ajouter une étoile "*" avant son nom. Que provoque en C la ligne suivante ?

```
printf("Contenu de la variable pointée via *p_i: %d\n",*p_i);
```

L'affichage du contenu présent à l'adresse pointée, donc le contenu de la variable pointée :

```
Contenu de la variable pointée via *p_i: 123456
```

Le fait d'accéder au contenu de la variable pointée s'appelle déréférencer un pointeur.

Avec ce système, il est possible de manipuler les variables et les pointeurs dans tous les sens. Si vous modifiez le contenu de la variable pointée, l'adresse du pointeur n'est pas modifiée, mais son déréférencement affichera la nouvelle valeur. Réciproquement, vous pouvez modifier la valeur de la variable pointée en passant par son pointeur avec l'étoile devant.

```
/* modification de la valeur de la variable pointée par p_i */
*p_i=987654;
printf("i contient maintenant %d\n",i);
```

Le résultat est que vous venez de modifier le contenu de la zone mémoire à l'adresse pointée par p_i, qui est celle de i. Vous venez donc de modifier la valeur de i en passant par un pointeur.

```
i contient maintenant 987654
```

c. Applications

Les applications sont nombreuses :

- En C une fonction ne sait pas retourner directement un tableau ou un enregistrement structuré. Elle doit retourner l'adresse de celle-ci et donc son résultat sera placé dans un pointeur adéquat.

- Dans le chapitre sur les sous-programmes, vous avez vu qu'il est possible de passer une variable en Sortie (S) ou Entrée/Sortie (ES) à une procédure. D'après vous, quel est le mécanisme utilisé ? Le pointeur, bien entendu : vous passez l'adresse de la variable en paramètre et le sous-programme va modifier le contenu de la mémoire à cette adresse via un pointeur.

- Les pointeurs ouvrent la voie à l'utilisation de mécanismes complexes. Notamment, vous pouvez créer une liste d'enregistrements ordonnés : un enregistrement contient une valeur, puis un pointeur vers l'enregistrement suivant et ainsi de suite…

- Dans un langage bas niveau comme le C, les tableaux se manipulent très facilement via des pointeurs, puisqu'il est possible de faire des calculs sur les adresses : +1 va au contenu de l'adresse suivante et ainsi de suite.

Voici un simple exemple en C d'une fonction qui doit modifier le contenu d'une variable en passant tout d'abord par une valeur de retour, puis par un pointeur.

La première fonction est très classique et ressemble beaucoup à ce qui existe en PHP.

```
long modif(long var,long n)
{
        var=n;
        return var;
}
```

Elle s'utilise ainsi et n'amène pas de remarque particulière :

```
i=modif(i,1000);
printf("i contient maintenant %d\n",i);
```

La seconde fonction est modifiée pour prendre une adresse en paramètre :

```
void modif2(long *var, long n)
{
        *var=n;
}
```

Le premier paramètre est un pointeur sur une variable de type long. La fonction doit recevoir l'adresse d'une variable de type long, comme la variable i de l'exemple. Vous devez l'appeler comme ceci :

```
modif2(&i,20000);
printf("i contient maintenant %d\n",i);
```

C'est l'adresse de i que vous passez comme premier paramètre de la fonction. Dans la fonction modif2, var va contenir l'adresse de i, va modifier le contenu de cette adresse en y plaçant la valeur du second paramètre. Au retour, la valeur de i est donc modifiée.

```
i contient maintenant 20000
```

C'est exactement le fonctionnement des sous-programmes de type procédure du chapitre Les sous-programmes.

C'est avec les enregistrements de type structuré et surtout les tableaux que les pointeurs montrent toute leur puissance. Un tableau est une liste d'éléments contigus en mémoire. Si vous récupérez l'adresse du premier élément, vous pouvez accédez aux n éléments suivants situés aux n adresses suivantes à l'aide d'un pointeur.

Prenez une chaîne "bonjour". Chaque caractère occupe un octet et en mémoire la chaîne se termine par un caractère nul. En C, une chaîne de caractères est en fait un tableau de caractères, l'élément d'indice 0 contenant le premier caractère, l'élément d'indice 1 le second, etc, jusqu'au dernier élément contenant le caractère nul de fin de chaîne. Placez un pointeur sur le premier élément du tableau. Si vous incrémentez le pointeur de 1, vous vous déplacez de la longueur d'un caractère en mémoire, vous vous trouvez sur le second caractère et ainsi de suite.



Ajouter 1 à un pointeur ne déplace pas forcément l'adresse de un octet en mémoire, mais de la longueur du type du pointeur. Si vous ajoutez 1 à un pointeur de type long, vous ajoutez quatre octets à l'adresse.

À l'aide de ce principe, il devient très simple de calculer la longueur d'une chaîne de caractères en C : tant que la valeur contenue à l'adresse du pointeur n'est pas nulle, vous incrémentez de 1 le pointeur. Ce qui donne :

```
char chaine[]="bonjour"; /* un tableau de caractères, une chaîne */
char *p_c; /*un pointeur de type caractère */
int n; /* va contenir le nombre de caractères */
/* le pointeur p_c contient l'adresse du premier élément */
p_c=&chaine[0];
/* tant que *p_c ne contient pas \0 (nul) */
while(*p_c!='\0')
{
p_c++; /* on décale d'un char l'adresse */
n++; /* on incrémente le compteur */
}
printf("Longueur: %d\n",n);
```

La boucle est extrêmement détaillée mais il est possible de faire bien plus court !

```
for(n=0;*p_c!='\0';p_c++) n++;
```

En C, chaque élément d'un tableau est en fait un pointeur sur l'adresse mémoire contenant cet élément. Ce qui veut dire que chaine contient l'adresse du premier élément et que *chaine contient le premier élément, chaine+1 l'adresse du second élément et *(chaine+1) son contenu, etc. Aussi p_c=&chaine[0] aurait pu s'écrire p_c=chaine.

Voyez-vous maintenant l'intérêt des pointeurs ? Bien que cette notion soit assez complexe, elle permet de simplifier fortement les traitements sur les tableaux, les passages et la récupération de valeurs et structures complexes.

3. Notation algorithmique

a. Déclarer et utiliser les pointeurs

L'algorithmique autorise bien entendu l'utilisation des pointeurs. Vous déclarez un pointeur comme n'importe quelle variable et au même endroit, sous le mot-clé VAR.

```
nom : pointeur sur type pointé
```

Par convention, les pointeurs commencent par la lettre p. Ce n'est pas une obligation mais vous vous y retrouverez beaucoup mieux si vous suivez cette recommandation. Comme vu précédemment, le type pointé doit être du même type que la variable pointée. Si vous créez un pointeur sur un entier, le pointeur sera de type entier : il pointe sur un entier.

```
Var
  txt :chaîne
  ptxt :pointeur sur chaîne
  cpt :entier
  pInt :pointeur sur entier
```

Le pointeur peut recevoir une adresse statique, c'est-à-dire une adresse en hexadécimal que vous rentrez vous-même. C'est une affectation directe. En pratique, cette méthode n'est que très rarement utilisée, sauf cas

spécifiques (si on sait qu'à telle ou telle adresse se trouve toujours la même donnée en toute circonstance) et vous préférerez passer l'adresse d'une variable connue.

```
pointeur←adresse de variable
```

Avec les variables du petit exemple, cela donne :

```
Début
  txt←"Hello World"
  ptxt←adresse de txt
  cpt←10
  pInt←adresse de cpt
```

Tout comme en C, vous utiliserez l'étoile pour accéder au contenu de la variable pointée, tant en lecture qu'en affectation. Suite du programme :

```
Afficher *ptxt
  *ptxt←"Salut tout le monde"
  Afficher txt
  *pInt←20
  cpt←cpt+1
  Afficher cpt
```

À la fin, que contiennent txt et cpt ? Respectivement "Salut tout le monde" et 21.

Il se peut qu'un pointeur n'ait pas encore reçu d'adresse et donc pointe sur nulle part. C'est embêtant car c'est le plantage assuré si vous tentez de l'utiliser. Pour éviter ce problème, vous lui affecterez une valeur générique, qui ne représente rien, mais qui pourra cependant être testée. C'est la valeur **NIL** (*Not Identified Link*). En C, c'est NULL et en PHP null. NIL est une sorte d'équivalent de zéro, mais qui est reconnue telle quelle (si vous comparez NIL et zéro, vous aurez un retour faux).

```
  pInt←NIL
Fin
```



Attention : c'est à vous de tester si un pointeur est positionné sur NIL avant de l'utiliser ! Un accès à un pointeur contenant NIL provoquera une erreur.

Dernier point, mais ceci devrait vous sembler évident, vous avez le droit de créer des pointeurs sur n'importe quel type de variable, y compris des enregistrements. C'est même l'un des piliers de l'utilisation des pointeurs. La suite du chapitre y fera fortement appel.

```
Type
  Structure tarticle
    ref:chaîne
    libelle:chaine
    prix:réel
  FinStruct
Var
  art :tarticle
  pArt :pointeur sur tarticle
Début
  art.ref←"ref01001"
  pArt←adresse de art
  Afficher (*pArt).ref // on trouve aussi la notation pArt→ref
Fin
```

En commentaire, il est indiqué que la notation pArt→ref est aussi utilisée, avec une flèche indiquant qu'on pointe sur l'enregistrement ref de la structure pointée. Cette notation est issue des langages C et C++ qui font la différence entre une variable structurée (utilisation du point pour l'accès aux champs) et un pointeur sur une variable structurée (utilisation de la flèche). Dans le doute, vous pouvez aussi procéder ainsi :

```
Afficher pArt→ref
Afficher (*pArt).ref
```

Les deux syntaxes sont équivalentes car rappelez-vous que l'étoile déréference le pointeur : on récupère la valeur de la variable pointée et donc ici l'équivalent de la variable art originale.

b. Allocation dynamique

Jusqu'à présent les pointeurs recevaient l'adresse d'une variable qui existait déjà par affectation. Il est aussi possible de réserver un emplacement mémoire pour une donnée pointée directement. Dans ce principe, vous pouvez créer un pointeur sur un entier par exemple et réserver un espace mémoire qui contiendra cet entier, sur lequel la variable pointeur pointera. C'est le principe de l'allocation dynamique de mémoire. Il vous faut employer la syntaxe suivante :

```
pointeur←nouveau type
```

Le type doit bien entendu être celui de la valeur qui sera contenue à l'emplacement mémoire réservé. Après cette instruction, le pointeur reçoit l'adresse mémoire de la zone réservée. En cas d'échec (plus de mémoire disponible par exemple) il reçoit la valeur NIL.

Dans l'exemple suivant, un pointeur sur un entier est déclaré. Voulant placer une valeur entière dans la zone mémoire pointée, il faut d'abord réserver l'emplacement nécessaire. Puis via l'utilisation de l'étoile devant le nom du pointeur, on y place un entier.

```
Programme alloc
Var
  pEntier :pointeur sur entier
Début
  pEntier←nouveau Entier
  *pEntier←12345
  Afficher *pEntier
Fin
```

Dans la plupart des langages disposant de pointeurs, il est possible de préciser la taille de la mémoire allouée, par exemple allouer un espace pour dix entiers. Dans ce cas, c'est l'équivalent d'un tableau d'entiers et l'adresse retournée sera celle du premier entier. Ajouter 1 au pointeur décalera celui-ci d'un élément. Cette syntaxe n'est pas utilisée en algorithmique où on préfère allouer la mémoire élément par élément, quitte à les chaîner ensuite.

Quand vous allouez dynamiquement de la mémoire, elle reste occupée tout le temps de l'existence du pointeur. Sans rien d'autre, la mémoire est récupérée uniquement à la sortie du programme. Il est aussi facile d'allouer de la mémoire que de la libérer, ou de la désallouer (un barbarisme bien utile) à volonté : dès que le ou les pointeurs ne sont plus utiles, on libère la mémoire associée, c'est cela de gagné. Pour ceci vous utiliserez la syntaxe suivante :

```
Libérer pointeur
```

Quand vous libérez le pointeur, vous libérez la zone mémoire sur laquelle il pointait, zone qui redevient disponible pour toute autre utilisation. Attention ! Si vous avez conservé dans un autre pointeur l'adresse de cette zone et que vous l'avez désallouée, ce pointeur pointe sur une zone éventuellement réaffectée à autre chose. Y accéder risque de fournir une valeur arbitraire, y écrire risque d'occasionner des problèmes, voire des plantages. Le mieux est de replacer une valeur NIL après la libération et de penser à tester le pointeur avant de l'utiliser.

Ne déréférencez jamais un pointeur dont la zone mémoire a été libérée. C'est une faute malheureusement courante. Dans de très gros programmes, le programmeur oublie parfois de tester la valeur du pointeur avant d'y accéder, provoquant une fuite mémoire aux conséquences souvent lourdes.

```
Programme libere
Var
  pEntier :pointeur sur entier
Début
  pEntier←nouveau Entier
  /* suite du programme */
  ...
  libérer pEntier
  pEntier←NIL
Fin
```

4. PHP et les références

a. Différences entre le C et PHP

Le langage C est le roi des pointeurs. À force d'en parler, on en oublierait presque PHP. Du fait de l'utilisation d'un interpréteur, PHP ne connaît pas les pointeurs au sens propre. Il est cependant possible qu'une variable référence l'autre. Cela ressemble un peu aux pointeurs. Cependant il existe des différences essentielles :

- Un pointeur contient l'adresse réelle en mémoire d'une variable. Une référence PHP, appelée handle (poignée) en anglais, ne contient qu'une information "virtuelle" permettant d'accéder à la ressource et fournie par l'interpréteur.
- Les manipulations de pointeurs peuvent vite devenir très complexes en C/C++ : risque de dépassement des adresses, allocations mémoires compliquées, risque de confondre les types, etc. Il n'y a pas de risque de ce genre en PHP où la mémoire est gérée par l'interpréteur.
- Le seul opérateur présent en PHP est le "&" permettant de référencer une variable.

Ne croyez pas que l'absence des pointeurs en PHP soit une limitation, c'est même tout le contraire. Tout d'abord PHP est un langage évolué de haut niveau dont le rôle n'est pas la manipulation bas niveau d'adresses physiques. De ce fait, le programmeur est entièrement débarrassé de toute la gestion de ces adresses puisque la machine virtuelle s'occupe de tout. C'est une sorte de C++ débarrassé de toute sa complexité. Le développement en PHP est donc très simplifié en se concentrant sur les fonctionnalités.

b. Les références

En PHP, toute variable peut être référencée et en référencer une autre. L'opérateur "&" devant un nom de variable permet de récupérer sa référence. Si vous tentez de l'afficher, vous n'obtiendrez pas d'adresse mémoire mais son nom de référence établi par l'interpréteur PHP. PHP maintient une table des ressources (variables par exemple) utilisées dans le programme. Chaque ressource est identifiée par une valeur, une entrée. C'est cette valeur qui référence la variable. Dans l'exemple suivant, $copie reçoit la référence du tableau $t. Les deux variables référencent donc la même chose : le tableau. On accède à celui-ci par l'une ou l'autre variable, puisqu'elles référencent le même tableau. De même, la fonction tableau reçoit comme paramètre la référence de la variable passée en paramètre. La modification est directement effectuée sur le tableau passé en paramètre.

En PHP toutes les valeurs sont passées en copie SAUF pour les objets, qui le sont toujours par référence et ceci depuis PHP5.

```
<html>
  <head><meta/>
    <title>Références explicites</title>
  </head>
  <body>
  <?php

  function tableau(&$tab)
  {
    $tab[1]=12345;
  }

  $t=array(2,7,9,10,11,14,17,18,20,22);
  $copie=&$t;

  echo "${t[2]}<br />\n";
  $copie[2]=5;

  echo "${t[2]}<br />\n";

  tableau($t);

  echo "${t[1]}<br />\n";

?>
  </body>
</html>
```

Vous reconnaissez l'un des exemples du chapitre "Les tableaux et structures". Il lui a été ajouté une fonction "tableau" qui prend comme paramètre un tableau d'entiers. Le tableau t est passé en paramètre de la fonction tableau. Dans cette fonction, le paramètre est modifié : l'indice 1 du tableau tab reçoit la nouvelle valeur 12345. Après la fin de la fonction, le programme réaffiche le contenu de l'indice 1 du tableau passé en paramètre : il contient 12345.

En PHP, tous les objets sont affectés et passés en référence aux fonctions. Par contre, tous les types simples (si on peut parler de types en PHP, parlez plutôt de scalaires) sont affectés et passés en copie. Vous pouvez forcer le passage en référence en ajoutant un & devant la variable dans la déclaration de la fonction ou l'affectation. C'est à vous de faire très attention lorsque vous modifiez le contenu de ces types de variables dans la fonction.

Il est possible de déclarer certains types classiques comme des objets pour faire réagir PHP comme Java, notamment avec les tableaux que Java considère comme des objets (Java considère tous les types comme des objets, tout ce qui est déclaré avec new). En PHP, il est possible de convertir un tableau en objet. Ainsi, il sera automatiquement affecté et passé en paramètre comme référence sans utiliser de & préalable.

```
<html>
  <head><meta/>
    <title>Référence avec ArrayObject</title>
  </head>
  <body>
  <?php

  function tableau($tab)
  {
    $tab[1]=12345;
  }

  $t=array(2,7,9,10,11,14,17,18,20,22);
  $o_t=new ArrayObject($t);

  $copie=$o_t;

  echo "${o_t[2]}<br />\n";
  $copie[2]=5;

  echo "${o_t[2]}<br />\n";

  tableau($o_t);

  echo "${o_t[1]}<br />\n";
?>
  </body>
</html>
```

Les références fonctionnent aussi sur des types quelconques.

```
<html>
  <head><meta/>
    <title>Référence sur types quelconques</title>
  </head>
  <body>
  <?php

  function modif_int($var)
  {
    $var=10;
    return $var;
  }

  function modif_int2(&$var)
  {
    $var=20;
  }

  $v=0;

  // aucun effet
  modif_int($v);
  echo "$v<br />\n";

  // par retour de valeur
  $v=modif_int($v);
```

```
  echo "$v<br />\n";

  // par reference
  modif_int2($v);
  echo "$v<br />\n";

?>
  </body>
</html>
```

c. Références sur structures

Dans la suite, vous allez avoir à manipuler des pointeurs sur des structures. C'est extrêmement simple car dans PHP toutes les soi-disant structures sont en fait des objets jusqu'à présent volontairement tronqués de la plupart de leurs fonctionnalités. Or comme tout objet, la variable qui reçoit un objet en reçoit une référence. Vous avez déjà utilisé ce principe sans le savoir, toujours dans le chapitre Les tableaux et structures, avec les structures tfabricant et tarticle.

Cet exemple est bien plus intéressant qu'il n'y paraît, vous le voyez probablement maintenant avec un œil neuf, puisque chaque déclaration d'une variable enregistrement de ces types est en fait une référence sur l'enregistrement :

- $article est une référence sur une structure (un objet) de type tarticle.
- $article contenant une variable de type tfabricant, on crée dedans une référence fab sur un objet de ce type.
- $art2 de type tarticle reçoit la référence de article. Ils référencent le même enregistrement.
- Le contenu des champs de $art2 est affiché, c'est le même que article. Si vous modifiez les champs, la modification se répercute sur toutes les références sur cet enregistrement : c'est le même pour tous.

```
<html>
  <head><meta/>
    <title>Référence structures</title>
  </head>
  <body>
  <?php

  class tfabricant {
    public $ref;
    public $nom;
    public $adresse;
    public $tel;
}
  class tarticle {
    public $ref;
    public $libelle;
    public $prix;
    public $fab;
  }


  $article=new tarticle();
  $article->ref="Art001_01";
  $article->fab=new tfabricant();
  $article->fab->ref="Fab1234";

  $art2=$article;

  echo "$art2->ref<br />\n";
  echo "{$art2->fab->ref}<br />\n";

?>
  </body>
</html>
```

d. Le piège en PHP

En PHP il faut faire très attention : quand vous passez une référence en paramètre, vous pouvez modifier le contenu de la référence et ce contenu sera modifié directement dans la zone mémoire, c'est parfait. Mais vous ne pouvez pas modifier la référence elle-même ! Si vous modifiez au sein d'une méthode la valeur de la référence elle-même, elle retrouve sa valeur initiale en sortie. Par exemple :

```
class element {
      public $valeur;
}

function modif($p1,$p2) {
      $p1=$p2;
}
```

Si dans le programme principal vous avez :

```
$p1=new element();
$p2=new element();

$p1->valeur=10;
$p2->valeur=15;

modif($p1,$p2) ;

echo $p1->valeur ;
```

La sortie sera 10 ! En effet dans la méthode modif(), p1 contient bien la référence vers l'emplacement mémoire de l'objet p1, mais la variable p1 est elle-même locale à la méthode ! Donc en affectant p2 à p1, p1 reçoit la référence de p2 mais la nouvelle valeur est perdue à la sortie de modif(). Il faut donc procéder ainsi :

```
function modif($p1, $p2) {
  $p1->valeur=$p2->valeur;
}
```

Cette fois, c'est le contenu des valeurs dans la zone mémoire référencée qui est affecté et vous obtenez le résultat attendu.

e. La valeur null

Il se peut que vous n'ayez pas besoin tout de suite de créer une référence sur un enregistrement, mais que vous vouliez éviter d'y accéder par inadvertance. Dans ce cas, au lieu de créer une référence avec new, vous pouvez lui affecter une valeur appelée **null**. Cette valeur signifie que la variable a été déclarée mais n'a pas d'objet instancié (terme expliqué dans le chapitre suivant) : vous ne lui avez pas encore affecté d'enregistrement. Ainsi, vous pouvez déjà tester la variable elle-même avant de tester les champs qu'elle contient.

Dans l'exemple suivant modifié, tarticle contient toujours une variable de type tfabricant. Mais elle ne recevra un enregistrement que plus tard dans le programme. En attendant, le champ fab reçoit une valeur nulle, signifiant que l'article n'a pas encore de fabricant référencé.

```
Html>
  <head><meta/>
    <title>Référence structures 2</title>
  </head>
  <body>
  <?php

  class tfabricant {
    public $ref;
  }

  class tarticle {
    public $ref;
    public $fab=null;
  }

  $article=new tarticle();
```

```
  $article->ref="Art001_01";

  if($article->fab!=null)
    echo "{$article->fab->ref}<br />\n";
  else
    echo "Pas de fabricant pour cet article<br />\n";

?>
  </body>
</html>
```

Les listes chaînées

1. Listes chaînées simples

a. Principe

Dans la vie quotidienne, une liste revêt plusieurs formes : une liste de courses, de tâches à effectuer, un index, un glossaire, une collection de dvds, de musiques, etc. Ces listes sont composées d'éléments individuels, liés les uns aux autres par leur type ou l'ordre que vous voulez leur donner. Pour passer d'un élément à un autre, vous descendez dans la liste dans l'ordre que vous lui avez donné.

Comment se représenter une liste, par définition linéaire, en programmation ? Vous connaissez au moins un moyen : les tableaux. Dans un tableau, vous pouvez stocker n éléments et l'ordre peut être représenté par l'indice du tableau.

Connaissez-vous un autre moyen de stocker des éléments ? Les enregistrements de types structurés le permettent et en plus vous pouvez y stocker bien plus de détails. Vous pouvez aussi créer des tableaux d'enregistrements, donc leur donner un certain ordre.

L'utilisation des tableaux pose cependant parfois des problèmes un peu complexes. Vous l'avez déjà remarqué avec les méthodes de tris.

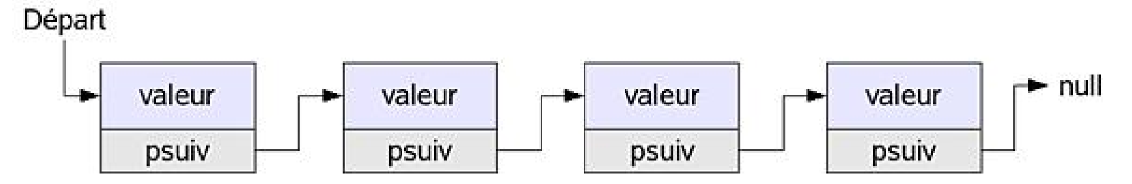
- Comment insérer un nouvel enregistrement en début de tableau ? Il n'y a pas d'indices négatifs…
- Comment insérer un nouvel enregistrement en fin de tableau ? Si l'algorithmique propose un redimensionnement dynamique, les langages PHP l'autorisent.
- Comment insérer un élément au milieu du tableau ? Faut-il décaler tous les éléments pour placer le nouveau à l'endroit donné ? Si oui, le tableau risque de déborder.
- Et si vous supprimez un enregistrement, allez-vous de nouveau décaler le tableau pour boucher le trou ? Ou trouver une parade pour passer par-dessus ?

Vous pouvez vous arranger pour tout programmer afin de tout faire marcher avec les tableaux. C'est parfaitement possible. Mais est-ce vraiment raisonnable ? Est-ce de la programmation efficace ? Rappelez-vous qu'au chapitre Introduction à l'algorithmique, vous avez appris qu'il faut être économe en ressources. Cette méthode est gourmande et compliquée. Il vous faut en trouver une autre plus simple et plus belle.

En fait, encore une fois, vous connaissez tous les principes de base de cette nouvelle méthode. Voici quelques éléments :

- Un enregistrement peut contenir un autre enregistrement.
- Cet autre enregistrement peut être du même type structuré.
- L'enregistrement peut aussi contenir un pointeur vers un autre enregistrement du même type.
- En notation Java, un enregistrement est un objet et dans un objet, on peut référencer un autre objet du même type.
- On obtient ainsi une cascade d'enregistrements qui se suivent les uns les autres. Pour accéder au suivant, il suffit d'accéder à la référence de cet enregistrement dans l'enregistrement actuel.
- Chaque enregistrement dispose d'un pointeur ou référence vers le suivant.
- Les enregistrements sont donc chaînés les uns aux autres, c'est une liste chaînée d'enregistrements.

Le principe peut être représenté par le schéma suivant. Chaque enregistrement est représenté par un cadre et contient une valeur et un pointeur appelé psuiv qui pointe sur l'enregistrement suivant de la liste.



Représentation logique d'une liste chaînée

Pour accéder à un élément donné de la liste vous partez toujours du premier élément. De là, connaissant l'adresse du suivant par le pointeur psuiv, vous passez successivement aux n suivants. Quand le pointeur ne pointe sur plus rien (null), c'est que l'enregistrement est le dernier.

Une liste chaînée de ce type est dite **unilatère** : l'accès aux éléments composant la liste est séquentiel (vous devez lire les n éléments précédents pour accéder à celui voulu) et unidirectionnel.

Chaque élément de la liste est un enregistrement. Cet enregistrement peut contenir autant de champs que vous le souhaitez, mais l'un d'eux sera un pointeur sur un enregistrement de même type. Quand vous ajouterez un deuxième élément, vous placerez son adresse dans le pointeur du premier et ainsi de suite. Chaque enregistrement final pointera sur la valeur NIL.

Pour commencer, voici un type structuré simple qui pourrait convenir. En fait la valeur pourrait être n'importe quoi et le type prendra juste un entier quelconque.

```
TYPES
  // Un élément de liste chaînée
Structure element
    valeur:entier
    pSuiv←NIL:pointeur sur element
FinStruct
```

Cette déclaration initiale du type structuré element contient un pointeur pSuiv (pour Pointeur sur Suivant) sur une structure du même type. Par défaut il est initialisé à la valeur NIL : il n'y a pas encore d'enregistrement suivant. Que pouvez-vous faire de cette structure ? Tout d'abord vous déplacer d'un élément à un autre de la liste. Pour cela, il suffit de partir du premier élément et de récupérer, en boucle, le pointeur de l'élément suivant, jusqu'à tomber sur NIL.

Quelles sont les opérations élémentaires possibles sur une liste ?

- **Créer une liste** : c'est créer le premier enregistrement, celui de tête, qui permettra d'accéder aux autres.
- **Parcourir une liste** : c'est balayer tous les éléments, un par un, jusqu'au dernier.
- **Rechercher un élément** dans la liste : soit indiquer s'il existe, soit retourner un pointeur vers sa position.
- **Ajouter un élément** n'importe où dans la liste : au début, au milieu, à la fin. Une fonction d'ajout pourrait aussi reprendre la première opération de création de liste.
- **Supprimer un élément** n'importe où dans la liste.
- **Supprimer la liste**.

Toutes ces actions peuvent se faire au travers de sous-programmes, rendant l'usage de la liste beaucoup plus simple. Pour se rapprocher des langages fonctionnels, les sous-programmes devant le plus souvent retourner un pointeur vers un élément de la liste.

Une dernière chose : le premier élément de la liste est toujours le point d'entrée pour la plupart des fonctions. Étant donné qu'il sera représenté par la suite par un pointeur, ne perdez JAMAIS l'adresse de ce premier élément : il serait impossible de retrouver le début de la liste, d'autant plus qu'avec l'allocation dynamique de mémoire, il est plus que possible que les zones mémoires allouées à chaque élément ne soient pas contiguës.

 Conservez toujours l'adresse du premier enregistrement dans un pointeur prévu à cet effet dont vous ne modifierez pas la valeur tout au long du programme, sauf si vous supprimez le premier élément ou toute la liste.

---

b. Création

Pour créer une liste, il faut commencer par son premier élément. Le premier élément est un pointeur auquel vous allez allouer dynamiquement une zone mémoire. Il prendra la valeur que vous voulez et son pointeur d'enregistrement suivant recevra NIL. La fonction retourne un pointeur vers le premier élément de la nouvelle liste.

```
Fonction cree_liste():pointeur sur element
Var
  pTete:pointeur sur element
Début
  pTete←nouveau element
  pTete→pSuiv ← NIL // (*pTete).pSuiv←NIL
  Retoune pTete
FinFonc
```



Rappel : pTete−>pSuiv <− NIL se lit ainsi : (*pTete).pSuiv←NIL, c'est-à-dire que le champ pSuiv de l'enregistrement pointé par pTete reçoit la valeur NIL.

Cette fonction amène un premier commentaire. Dans le chapitre Les sous-programmes, vous avez appris la différence entre les variables locales et globales. Ici pTete est une variable locale, elle sera détruite à la fin de la fonction. Pourtant l'adresse qu'elle contient est retournée. C'est que le pointeur n'étant pas libéré, la zone mémoire allouée dynamiquement l'est pour toute la durée du programme. À la sortie de la fonction, la zone mémoire existe encore et donc son adresse est encore valide.

Remarquez que l'affectation de la valeur NIL à pSuiv n'est pas nécessaire car c'est sa valeur par défaut lors de la déclaration de l'enregistrement structuré.

Pour exploiter cette fonction, il suffit de déclarer un pointeur et lui affecter le résultat de celle-ci :

```
Programme liste1
Var
  pTete:pointeur sur element
Début
  pTete←cree_liste();
Fin
```

La fonction cree_liste est très simple. Peut-être pourriez-vous en profiter pour voir le mécanisme simple permettant d'ajouter des éléments les uns à la suite des autres. La fonction cree_liste2() modifiée va vous demander de saisir en boucle des valeurs qui seront ajoutées les unes après les autres en fin de liste. Pour ceci, vous aurez besoin de conserver à chaque fois trois informations :

- Le pointeur pTete de la tête de la liste, qui devra être retourné par la fonction.
- Le pointeur pEncours de l'élément actuel de la liste, de l'élément ajouté en fait.
- Le pointeur pPrec de l'élément précédent, dont le pointeur pSuiv devra recevoir l'adresse de l'élément en cours.

```
Fonction cree_liste2() :pointeur sur element
Var
  pTete, pEncours, pPrec :pointeurs sur element
  v :Entier
Début
  // 1er élément
  pTete←nouveau element
  Afficher "Valeur du premier élément ?"
  Saisir pTete→valeur

  // Le premier élément est le précédent de l'élément suivant
  pPrec←pTete
  Répéter
    Afficher "Valeur suivante (0=sortie) ?"
    Saisir v
    Si v<>0 Alors
      // Allocation du nouvel élément
      pEncours←nouveau element
      pEncours→valeur ← v
```

```
      // Chaînage : pEncours est le suivant de pPrec
      pPrec→pSuiv ← pEncours
      pPrec←pEncours
    FinSi
  Jusqu'à v=0

  // Fin de liste : pSuiv à NIL
  pPrec→pSuiv ← NIL
  Retourne pTete
FinFonc
```

c. Parcours de la liste

Le parcours de la liste est maintenant possible puisque la fonction précédente vous a permis de remplir quelques enregistrements. Comme toujours on part du pointeur de tête, puis on passe d'enregistrement en enregistrement jusqu'à rencontrer la valeur NIL.

La fonction parcours_liste reçoit comme argument le pointeur de tête. Une simple boucle va ensuite balayer toute la liste et afficher toutes les valeurs qui y sont contenues.

```
Fonction parcours_liste(pTete :pointeur sur element)
Var
  pEncours :pointeur sur élément
Début
  pEncours←pTete
  Tant que pEncours<>NIL Faire
    Afficher pEncours→valeur
    pEncours ← pEncours→pSuiv
  FinTantQue
FinFonc
```

d. Recherche

Deux types de sous-programmes sont possibles : le premier détermine si l'élément existe dans la liste et retourne un booléen, vrai ou faux, selon que l'élément est trouvé ou non. C'est donc une fonction. Le second retourne l'adresse de l'élément trouvé et l'adresse de l'élément précédent, vous verrez pourquoi ensuite. C'est donc une procédure car une fonction ne peut retourner deux valeurs. Mais pourquoi ne pas faire d'une pierre deux coups, c'est-à-dire un sous-programme qui va à la fois retourner vrai ou faux, mais aussi des pointeurs sur l'élément courant et précédent ?

C'est possible car vous pouvez passer des pointeurs comme arguments de fonctions, modifier l'adresse sur laquelle ils pointent et retourner tout de même un booléen. En fait, il est quasiment inutile de retourner un booléen car de toute façon si l'élément n'est pas trouvé, pEncours vaudra NIL en sortie de fonction.

La procédure recherche_liste prend quatre paramètres :

- La valeur v recherchée.
- Un pointeur pTete sur la tête de la liste.
- Un pointeur pPrec sur l'élément précédent, celui trouvé.
- Un pointeur pEncours sur l'élément trouvé.
- Un booléen vrai ou faux.
- Si l'élément est trouvé, pPrec pointe sur celui d'avant, pEncours sur l'élément trouvé.
- Si l'élément est absent, pEncours vaut NIL et pPrec pointe sur le dernier élément de la liste.
- Si pPrec vaut NIL, l'élément recherché est le premier de la liste.

```
Procédure recherche_liste (E:v:entier, E:pTete, ES : pPrec,pEncours
:pointeurs sur element,S :trouve :booléen)
```

```
Var
  trouve :booléen
Début
  trouve←FAUX
  pPrec←NIL
  pEncours←pTete
  Tant que pEncours<>NIL ET pEncours→valeur<>v Faire
    pPrec←pEncours
    pEncours ← pEncours→pSuiv
  FinTantQue
  Si pEncours<>NIL ET pEncours→valeur=v Alors
    trouve←VRAI
  FinSI
FinProc
```

e. Ajout d’un élément

Pour l’ajout d’un élément dans la liste, trois cas de figure peuvent se présenter, nécessitant trois traitements différents :

- L’ajout d’un élément en début de liste.
- L’ajout d’un élément en milieu de liste.
- L’ajout d’un élément en fin de liste.

Dans les trois cas, le chaînage est modifié. Il est certes possible de créer un gros sous-programme qui gère les trois cas d’un coup, mais il est profitable de différencier ces trois traitements en trois sous-programmes indépendants. Il sera toujours temps ensuite de créer un sous-programme fédérateur qui gèrera tous les cas.

Dans tous les cas suivants, l’adresse de l’élément à ajouter, que vous aurez déjà remplie avec la bonne valeur, sera représentée par le pointeur pNouveau.

Ajout en début de liste

Deux cas sont possibles : l’ajout d’un élément dans une liste vide, auquel cas il s’agit du premier enregistrement et l’ajout d’un élément en première position de la liste.

Dans le premier cas, il s’agit de faire en sorte que l’élément à ajouter soit le premier, donc que pNouveau devienne l’élément de tête de la liste, sans élément suivant. Vous passez à la procédure le pointeur sur l’enregistrement et le pointeur de tête.

```
Procédure ajout_unique(E :pNouveau, ES :pTete: pointeurs sur element)
Début
  pNouveau→pSuiv ← NIL
  pTete←pNouveau
FinProc
```

La seconde procédure ajoute le nouvel élément en tête de liste, sachant que la liste contient déjà au moins un élément. C’est un cas très simple où le nouvel élément reçoit comme élément suivant celui de tête. Pour ce dernier, rien ne change.

```
Procédure ajout_début(E :pNouveau, ES :pTete :pointeurs sur element)
Début
  pNouveau→pSuiv ← pTete
  pTete←pNouveau
FinProc
```

Ajout en fin de liste

C’est encore un cas très simple. Le rajout d’un élément en fin de liste nécessite seulement de connaître l’adresse du dernier élément actuel. Le pointeur pSuiv de ce dernier pointera sur le nouvel élément et le pSuiv du nouvel élément recevra NIL.

Si vous reprenez les procédures de recherche et de parcours de la liste chaînée, à la fin de la liste pEncours vaut NIL et pPrec contient l’adresse du dernier enregistrement. La procédure ajout_fin() reçoit deux paramètres : pNouveau et pPrec.

```
Procédure ajout_fin(ES : pNouveau, pPrec :pointeurs sur element)
Début
  pPrec→pSuiv ← pNouveau
  pNouveau→pSuiv ← NIL
FinProc
```

Ajout en milieu de liste

Au final, aucun ajout n'aura été bien compliqué, puisque là encore vous disposez de tout le nécessaire. Pour ajouter un élément entre deux autres éléments d'une liste, vous devez connaître l'adresse de l'élément précédent et l'adresse de l'élément courant, sachant que le nouvel élément sera inséré entre les deux. La procédure ajout_milieu () reçoit donc trois arguments : le nouvel élément pNouveau, l'élément précédent pPrec et l'élément actuel pEncours.

```
Procédure ajout_milieu(ES :pNouveau, pPrec, pEncours) :pointeurs
sur element)
Début
  pPrec←pSuiv → pNouveau
  pNouveau←pSuiv → pEncours
FinProc
```

Généralisation

Le but est de généraliser les ajouts en un grand sous-programme unifié. Pour cela, il faut savoir où placer l'élément à ajouter. L'algorithme de recherche d'un élément retourne deux pointeurs : celui de l'élément trouvé et de l'élément précédent. Supposez que vous souhaitez insérer votre nouvel élément juste avant l'élément recherché, cela devient plutôt simple. Quatre cas de figure se présentent :

- pPrec contient NIL (pas d'élément avant) et pEncours contient NIL : il n'y a aucun élément dans la liste, le nouveau sera le premier et seul élément.
- pPrec contient NIL (pas d'élément avant) et pEncours pointe sur l'élément trouvé qui est le premier : le nouveau se place avant, en premier.
- pPrec pointe sur un élément précédent et pEncours pointe sur l'élément trouvé, le nouveau se place au milieu des deux.
- pPrec pointe sur un élément précédent et pEncours contient NIL (la recherche est arrivée au bout, élément trouvé ou non), le nouveau se place en dernier.

Quatre cas de figures qui correspondent aux quatre sous-programmes déjà créés ci-dessus ! L'algorithme devient simple : il suffit d'appeler la bonne procédure selon les valeurs de pPrec et de pEncours. La procédure ajout_element prend trois valeurs : la valeur recherchée, la valeur à insérer avant et le pointeur de tête de la liste.

```
Procédure ajout_element(E:vrech,vnouveau :entiers, ES :pTete
:pointeur sur element)
Var
  pPrec, pEncours, pNouveau : pointeurs sur element
Début
  pNouveau←nouveau element ;
  pNouveau→valeur ← vnouveau
  recherche_liste(vrech,pTete,pPrec,pEncours)
  Si pPrec=NIL Alors
    Si pEncours=NIL Alors
      ajout_unique(pNouveau, pTete)
    Sinon
      ajout_début(pNouveau, pTete)
    FinSI
  Sinon
    Si pEncours=NIL Alors
      ajout_fin(pNouveau, pPrec)
    Sinon
      ajout_milieu(pNouveau, pPrec, pEncours)
    FinSi
  FinSi
FinProc
```

Simplification

Les traitements sont ici très détaillés. Cependant, analysez les procédures ajout_premier() et ajout_début(). Que remarquez-vous ? Elles se ressemblent, d'autant plus que dans ajout_début(), pTete contient déjà NIL s'il n'y pas d'enregistrements. Donc les deux procédures sont identiques : ajout_debut() remplace ajout_premier().

Regardez maintenant ajout_fin() et ajout_milieu() : dans ajout_milieu, que reçoit pEncours->pSuiv si vous êtes en fin de liste ? La valeur NIL. Donc les deux sont identiques et ajout_mlilieu() peut remplacer ajout_fin(). La procédure ajout_element se trouve simplifiée ainsi.

Les deux autres procédures ne servent plus à rien.

```
Procédure ajout_element(vrech,vnouveau :entiers, pTete
:pointeur sur element)
  ...
  Si pPrec=NIL Alors
    ajout_début(pNouveau, pTete)
  Sinon
    ajout_milieu(pNouveau, pPrec, pEncours)
  FinSi
FinProc
```

f. Suppression d'un élément

L'ajout d'éléments est un grand pas en avant car vous connaissez et comprenez intégralement le principe inhérent aux listes chaînées. Pour supprimer un élément de cette liste, c'est quasiment la même chose, il faut juste recoller les morceaux et libérer la mémoire allouée à l'élément supprimé. Il y a quatre possibilités :

- Supprimer le seul élément de la liste ;
- Supprimer le premier élément de la liste ;
- Supprimer le dernier élément de la liste ;
- Supprimer un élément au milieu de la liste.

Il faut que l'élément à supprimer existe, donc tout d'abord le rechercher et connaître son adresse et celle de l'élément précédent. L'élément précédent verra son pointeur pSuiv prendre comme valeur le pointeur pSuiv de l'élément trouvé.

Les fonctions suivantes restructurent la liste pour lui redonner le bon chaînage. La libération de la mémoire occupée par l'élément à effacer sera effectuée selon le même modèle que la fonction fédératrice d'ajout dans une grande fonction de suppression.

Supprimer le seul élément

C'est facile, si l'élément est le seul, donc la tête de la liste, il suffit de placer son pointeur à NIL. Pas de précédent, pas de suivant, c'est terminé.

```
Procédure suppr_unique(ES :pTete :pointeur sur element)
Début
  pTete←NIL
FinProc
```

Supprimer le premier élément

C'est aussi simple : c'est l'élément suivant qui devient la tête de la liste.

```
Procédure suppr_premier(ES :pTete :pointeur sur element)
Début
  pTete ← pTete→pSuiv
FinProc
```

Supprimer le dernier élément

Toujours aussi simple : l'élément précédent reçoit NIL comme valeur de pointeur suivant.

```
Procédure suppr_dernier(E :pPrec :pointeur sur element)
Début
  pPrec→pSuiv ← NIL
FinProc
```

Supprimer un élément au milieu

Il faut raccorder l'élément précédent avec l'élément suivant.

```
Procédure suppr_milieu(E :pPrec,pEncours : pointeurs sur element)
Début
  pPrec→pSuiv ← pEncours→pSuiv
FinFonc
```

Simplification

Contrairement à l'ajout, vous pouvez tout de suite voir s'il est possible de simplifier avant d'aller plus loin, selon le même principe. Ainsi dans la fonction suppr_premier() pTete recevra NIL si l'élément supprimé est le seul car pTete->pSuiv vaut NIL.

Idem pour suppr_milieu() et suppr_dernier(). Dans suppr_milieu, pEncours->pSuiv vaut NIL si l'élément est le dernier.

Les fonctions suppr_unique() et suppr_dernier() sont donc inutiles, sauf pour l'exemple !

Généralisation

Tout d'abord vous devez trouver l'élément à supprimer. S'il n'y est pas, il n'y a rien à supprimer. Donc si pEncours, contenant l'élément trouvé, vaut NIL, il n'y a rien à faire. Ensuite, il y a deux cas de figure :

- pPrec vaut NIL : l'élément à supprimer est le premier (ou le seul).
- pPrec est différent de NIL, l'élément à supprimer est au milieu ou en fin de liste.

Ensuite, après avoir appelé la fonction adéquate, vous n'avez plus qu'à libérer la mémoire allouée pour l'élément et passer son pointeur à NIL.

La procédure suppr_element ne prend que deux arguments : la valeur de l'élément à supprimer et la tête de la liste.

```
Procédure suppr_element(E:vrech:entier,ES:pTete:pointeur sur element)
Var
  pPrec, pEncours :pointeurs sur element
Début
  recherche_liste(vrech,pTete,pPrec,pEncours:pointeurs sur elements)
  Si pEncours=NIL Alors
    Afficher "Element absent"
  Sinon
    Si pPrec=NIL Alors
      suppr_premier(pTete)
    Sinon
      suppr_milieu(pPrec,pEncours)
  FinSi
  Libérer pEncours
  pEncours←NIL
FinProc
```

g. Supprimer toute la liste

Pour supprimer tous les éléments de la liste, il suffit de supprimer tous les éléments jusqu'au dernier. Mais attention ! Ne supprimez pas un élément sans avoir auparavant conservé l'adresse de l'élément suivant ! Si vous ne l'avez pas fait, non seulement les éléments suivants sont perdus, mais la mémoire des éléments suivants ne pourra plus être libérée !

La fonction suppr_liste ne prend qu'un seul argument : la tête de la liste.

```
Fonction suppr_liste(pTete :pointeur sur element)
Var
  pEncours, pSuivant :pointeurs sur élément
Début
  pEncours←pTete
  Tant que pEncours<>NIL Faire
    pSuivant←pEncours→pSuiv
    Libérer pEncours
    pEncours←pSuivant
  FinTantQue
  pTete=NIL
FinFonc
```

h. Parcours récursif

Il est possible de remplacer le sous-programme itératif de parcours de la liste par une fonction récursive : le sous-programme s'appelle lui-même avec l'adresse de l'élément suivant tant que l'élément reçu en argument n'est pas NIL.

```
Fonction parcours_recursif(pEncours :pointeur sur element)
Début
  Si pEncours<>NIL Alors
    Afficher (*pEncours).valeur
    parcours_recursif(pEncours→pSuiv)
  FinSi
FinFonc
```

Cette fonction est appelée avec le pointeur de tête comme paramètre.

```
parcours_recursif(pTete)
```

2. L'implémentation en PHP

Voici le programme adapté au PHP. Il faut adapter quelques procédures pour qu'elles retournent une référence vers les divers éléments de la liste.

```
<?php

  class element {
    public $valeur;
    public $pSuiv=null;
  }

  // Creation de la tete
  function cree_liste() {
    $pTete=new element();
    $pTete->pSuiv=null;
    return $pTete;
  }

  // tete et suivants
  function cree_liste2() {
    // 1er élément
    pTete=new element();
    echo "1er élément ? ";
    $pTete->valeur=(int)(fgets(STDIN));

    $pPrec=$pTete;

    // elements suivants
    do {
      echo "Element suivant ? ";
      $v=(int)(fgets(STDIN));
      if($v!=0) {
```

```
      $pEncours=new element();
      $pEncours->valeur=$v;

      // chaînage
      $pPrec->pSuiv=$pEncours;

      $pPrec=$pEncours;
    }
  } while($v!=0);

  // Fin de liste
  $pPrec->pSuiv=null;
  return $pTete;
}

// parcours itératif
function parcours_liste($pTete) {

  $pEncours=$pTete;

  while($pEncours!=null) {
    echo $pEncours->valeur." ";
    $pEncours=$pEncours->pSuiv;
  }
  echo "\n";
}

// adapté pour PHP
function recherche_liste($v, $pTete) {
  $pPrec=null;
  $pEncours=$pTete;

  while($pEncours!=null && $pEncours->valeur!=$v) {
    $pPrec=$pEncours;
    $pEncours=$pEncours->pSuiv;
  }
  return $pPrec;
}

// Rajouté pour PHP
function existe_liste($v, $pTete) {
  $pPrec=null;
  $trouve=false;

  $pPrec=recherche_liste($v,$pTete);

  if($pPrec==null) {
    if($pTete!=null && $pTete->valeur==$v) $trouve=true;
  } else {
    if($pPrec->pSuiv!=null) $trouve=true;
  }
  return $trouve;
}

// Fonctions d'ajout
function ajout_debut($pNouveau, $pTete) {
  $pNouveau->pSuiv=$pTete;
  $pTete=$pNouveau;
  return $pTete;
}


function ajout_milieu($pNouveau, $pPrec, $pEncours) {
  $pPrec->pSuiv=$pNouveau;
  $pNouveau->pSuiv=$pEncours;
}

function ajout_element($vrech, $vnouveau, $pTete) {
  $pPrec=null;
```

```
  $pEncours=null;

  $pNouveau=new element();
  $pNouveau->valeur=$vnouveau;

  $pPrec=recherche_liste($vrech,$pTete);
  if($pPrec!=null) $pEncours=$pPrec->pSuiv;

  if($pPrec==null) $pTete=ajout_debut($pNouveau, $pTete);
  else ajout_milieu($pNouveau, $pPrec, $pEncours);

  return $pTete;
}

// Fonctions de suppression
function suppr_premier($pTete) {
  $pTete=$pTete->pSuiv;
  return $pTete;
}

function suppr_milieu($pPrec, $pEncours) {
  $pPrec->pSuiv=$pEncours->pSuiv;
}

function suppr_element($vrech, $pTete) {
  $pPrec=null;
  $pEncours=null;

  $pPrec=recherche_liste($vrech,$pTete);
  if($pPrec!=null) $pEncours=$pPrec->pSuiv;
  else $pEncours=$pTete;

  if($pEncours!=null) {
    if($pPrec==null) $pTete=suppr_premier($pTete);
    else suppr_milieu($pPrec,$pEncours);
  }
  $pEncours=null;
  return $pTete;
}

// supprimer la liste
function suppr_liste($pTete) {
  $pTete=null;
  return $pTete;
}

// parcours récursif
function parcours_recursif($pEncours) {
  if($pEncours!=null) {
    echo $pEncours->valeur." ";
    parcours_recursif($pEncours->pSuiv);
  } else echo "\n";
}

// programme principal

$pTete=cree_liste2();

parcours_liste($pTete);

echo "Saisir la valeur recherchee : ";
$v=(int)(fgets(STDIN));

if(existe_liste($v,$pTete)) echo "Trouvé !\n";
else echo "Absent !\n";

$pTete=ajout_element(2,15,$pTete);

parcours_liste($pTete);
```

```
  echo "Saisir la valeur à supprimer : ";
  $v=(int)(fgets(STDIN));
  $pTete=suppr_element($v,$pTete);

  parcours_recursif($pTete);

  suppr_liste($pTete);
?>
```

3. Autres exemples de listes

a. Listes circulaires

Une liste circulaire permet d'accéder à n'importe quel élément de la liste depuis n'importe quel autre élément sans passer par le pointeur de tête.

Pour mettre en place ce type de liste, il suffit de faire pointer l'élément suivant du dernier élément sur le pointeur de tête.

Dans une telle liste, les fonctions d'ajout et de suppression d'éléments sont simplifiées et correspondent aux fonctions d'ajout et de suppression au milieu. Il y a un petit problème pour la fonction de recherche qui ne s'arrête jamais. Il faut donc placer un drapeau pour arrêter la recherche quand on retombe sur l'élément de départ. Il suffit de stocker l'adresse de départ. Si vous retombez dessus, c'est que vous avez parcouru toute la liste.

b. Listes d'éléments triés

Dans ce type de liste, les éléments sont placés selon un ordre défini par vous-même au sein des valeurs contenues dans les éléments. Ainsi vous êtes assuré de respecter cet ordre lors du parcours de la liste.

Vous devez adapter la fonction de recherche pour faire respecter cet ordre.

c. Listes doublement chaînées

Dans une liste chaînée simple, le parcours ne s'effectue que dans un seul sens et depuis un élément vous ne pouvez pas retourner au précédent. Une première méthode consisterait à conserver l'adresse de chaque élément précédent, ce qui serait possible avec l'utilisation de fonctions récursives, mais ce serait très lourd.

L'autre possibilité consiste à utiliser des listes doublement chaînées : chaque élément ne contient non plus un seul pointeur, mais deux : un pour l'élément suivant, un autre pour l'élément précédent. Ainsi, vous pouvez vous déplacer dans les deux sens dans la liste, soit vers la queue (fin), soit vers la tête. Ces listes sont dites bilatères.

Les traitements doivent être adaptés en conséquence. Prenez le problème à l'envers : vous avez deux pointeurs à mettre à jour. Si vous savez le faire dans un sens (comme les listes simples) vous savez le faire dans l'autre, vous changez juste de sens. Il y a donc deux fois plus d'opérations de chaînage :

Chaînage avant

- pPrec→pSuiv ← pNouveau
- pNouveau→pSuiv ← pSuivant

Chaînage arrière

- pSuivant→pPrec ← pNouveau
- pNouveau→pPrec ← pPrec

Les listes bilatères peuvent aussi être circulaires et/ou triées.

d. Files et piles

Dans une file d'attente, dans un magasin, un cinéma, bref dans une queue, le premier arrivé est le premier servi. En anglais, cela se traduit par "First In, First Out", soit FIFO en abrégé.

Une file d'attente de type FIFO peut être représentée par une liste chaînée. Chaque nouvel élément est ajouté en fin de liste, tandis que les éléments sont traités les uns après les autres depuis la tête de la liste.

Quand vous faites la vaisselle, les assiettes sont empilées les unes sur les autres. Quand vous lavez les assiettes vous prenez celles du dessus en descendant au fur et à mesure. Si de nouvelles assiettes sales sont empilées, elles le sont sur le dessus de la pile.

Le principe est le même en informatique quand vous voulez traiter des éléments au fur et à mesure de leur arrivée : les derniers arrivés sont traités en premier. La pile peut être représentée par une liste chaînée : les nouveaux éléments sont systématiquement empilés en tête de liste et les éléments sont toujours traités depuis cette tête. Si les éléments arrivent plus vite que leur traitement, ceux arrivés en premier risquent d'être traités bien tard.

Les arbres

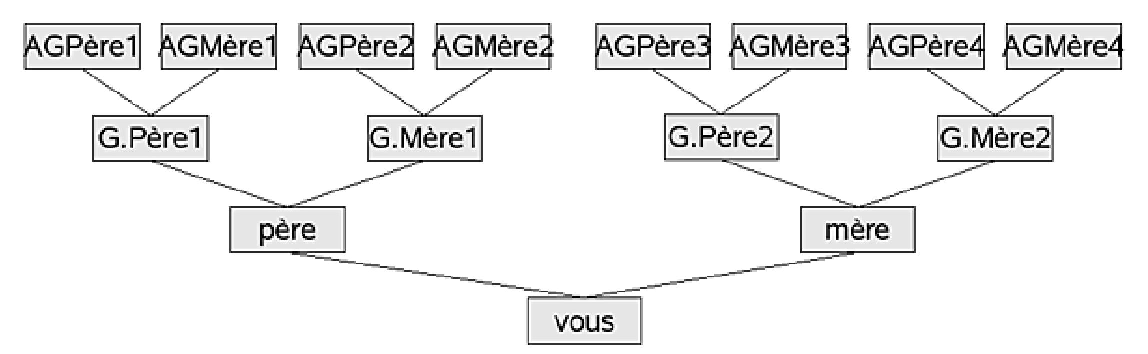
1. Principe

L'implémentation des arbres en PHP reprend des principes quasi-identiques aux listes vues précédemment. Cette fois, le code PHP ne vous sera pas fourni ! À vous d'implémenter ces algorithmes, ce n'est pas très difficile.

Dans la nature, les végétaux décrivent souvent des structures dites arborescentes. L'exemple le plus évocateur est l'arbre : le tronc se décompose en plusieurs branches, se décomposant elles-mêmes en branches plus petites et ainsi de suite jusqu'aux extrémités où poussent les feuilles.

Selon le cas, après les tableaux et les listes, vous pouvez vous aussi choisir de représenter l'organisation de vos données sous forme d'arborescence en programmation. La notion d'arborescence est très courante sur votre ordinateur personnel, de nombreuses informations sont représentées, directement ou indirectement sous forme d'arborescence : les dossiers des disques durs, la structure d'une page web, la structure d'un site web, la décomposition de l'exécution d'un programme et de ses appels aux sous-programmes, bref tout ce qui peut incorporer une notion de hiérarchie peut être représenté sous forme d'une arborescence.

L'exemple le plus simple à comprendre est la généalogie. On parle d'arbre généalogique. Partant de vous (1), vous placez tout d'abord vos parents (2), puis les parents de vos parents (4), puis les parents de ces derniers (8) et ainsi de suite. Tous sont reliés par leurs liens de parenté : vous avec vos parents, parents avec grands-parents et ainsi de suite. Le schéma part de vous, mais pourrait partir de vos arrières grands-parents, ayant x enfants, y petits-enfants, z arrières-petits enfants (dont vous), chaque individu étant le successeur de son parent et le prédécesseur de ses enfants.



Un arbre généalogique est un arbre binaire

Comment représenter un tel arbre généalogique en programmation ? Vous disposez comme souvent de plusieurs moyens, notamment avec les bases de données, mais connaissant les listes chaînées, vous devez penser qu'il existe un moyen ou un autre de s'en sortir directement avec quelques enregistrements et pointeurs. Vous avez raison.

Dans un arbre, chaque élément (membre de votre famille) dispose d'un père et d'une mère, qui ont eux-mêmes deux parents. Un élément peut donc être décrit par plusieurs informations mais deux seulement vont vous intéresser pour la suite : un élément individu pointe sur son père et sa mère. Un type structuré pouvant représenter un individu pourrait donc être :

```
Structure individu
  nom:chaîne
  pnom :chaîne
  ...
  pPère:pointeur sur individu
  pMère:pointeur sur individu
FinStruct
```

Contrairement aux listes chaînées simples ou doubles, il ne s'agit pas ici de définir une liste, file ou pile mais une notion de hiérarchie entre éléments pères et fils. Cette hiérarchie porte le nom d'arbre.

2. Définitions

a. Base

Un arbre est formé d'une racine qui est l'élément à la base de l'arbre et d'un nombre fini d'arbres qui lui sont raccordés appelés sous-arbres.

Chaque élément d'un arbre peut avoir plusieurs successeurs (vous avez deux parents) mais un seul prédécesseur. Seule la racine n'a pas de prédécesseur.

b. Terminologie

Les arbres utilisent une terminologie particulière qui reprend en gros celle de la nature et de la généalogie :

- Un **nœud** ou sommet est un élément quelconque de l'arbre. Dans un arbre généalogique, chaque individu représente un nœud ou sommet : il a plusieurs successeurs mais un seul prédécesseur.
- La **racine** est le premier élément de l'arbre, n'ayant pas de prédécesseur dans la hiérarchie.
- Une **feuille** ou nœud terminal, ou nœud final est un élément qui n'a pas de successeur.
- Un **nœud interne** est un nœud qui n'est ni racine, ni feuille, qui a donc un prédécesseur et des successeurs.
- Un **arc** relie deux nœuds.
- Une branche est le chemin qui relie la racine à une feuille.

Du côté du rapprochement généalogique légèrement sexiste, vous trouverez les termes suivants :

- Le **père** est le prédécesseur unique d'un nœud.
- Les **fils** sont les n successeurs d'un nœud.
- Les nœuds de père identique sont des **frères**.
- Le nœud le plus à gauche de l'arbre est **l'aîné**.

Un arbre peut être décrit horizontalement et verticalement.

c. Description horizontale

Horizontalement, un arbre **n-aire** est un arbre dont le nombre maximum de fils par nœud est n. Les fils sont regroupés par niveaux. Un **niveau** est l'ensemble des nœuds à égale distance de la racine. Le premier niveau est la racine, le deuxième les fils de la racine, le troisième les fils des fils et ainsi de suite. Quand chaque nœud d'un niveau a exactement n fils, le niveau est dit **saturé**.

Un arbre est **strictement complet** si tous les niveaux sont complets. Il est simplement **complet au sens large** si tous les niveaux intermédiaires sont complets mais qu'il manque des feuilles. Dans un arbre strictement complet, la racine et tous les nœuds internes ont exactement n fils, ni plus ni moins.

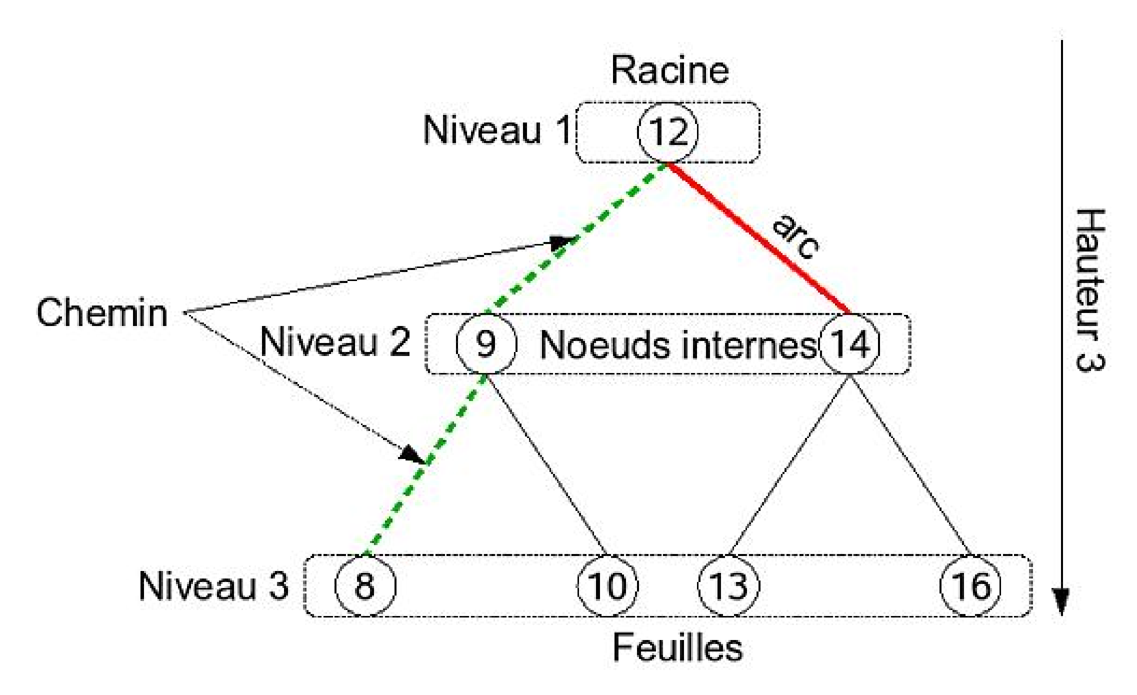
d. Description verticale

La hauteur d'un arbre est le nombre de nœuds du plus long chemin direct, la plus longue branche entre la racine et une feuille, racine et feuille inclues. Si l'arbre dispose d'une racine, de deux fils et qu'un des fils a une feuille, la hauteur de l'arbre est 3.

e. L'arbre binaire

Un arbre binaire est un arbre dont chaque nœud a au plus deux fils. Depuis la racine, l'arbre binaire est constitué de deux sous-arbres différenciés, le **sous-arbre droit** et le **sous-arbre gauche**.

Il existe des arbres à trois, quatre, n fils. Cependant seuls les arbres binaires seront abordés ici et notamment les arbres binaires ordonnés. Le schéma suivant montre un arbre binaire strictement complet de hauteur 3.



Arbre binaire ordonné strictement complet de hauteur 3

3. Parcours d'un arbre

Pour la suite, la structure d'un nœud d'un arbre sera la suivante, étant bien compris qu'à chaque nœud est associée une valeur (sinon l'arbre n'a aucun intérêt) et que bien que cette structure ressemble à celle d'un élément de liste doublement chaînée ce n'est plus pour une représentation linéaire mais hiérarchique.

```
Structure noeud
  valeur:entier
  pGauche:pointeur sur noeud
  pDroit:pointeur sur noeud
FinStruct
```

Chaque arbre binaire peut être décomposé en sous-arbres, un à gauche et un à droite. Mais bien souvent un sous-arbre peut lui-même être éclaté : chaque nœud ayant un ou des fils, contient un ou deux sous-arbres, un à droite et un à gauche. Le nœud 9 a deux sous-arbres : un à gauche vers le nœud 9, un à droite vers le nœud 10. Dans les fonctions de parcours suivantes, à chaque appel chaque nœud ne valant pas NIL est considéré comme la racine d'un arbre et les sous-arbres partant de ce nœud seront parcourus comme tels.

Le parcours complet d'un arbre consiste à parcourir tout l'arbre afin d'accéder à l'ensemble des nœuds de l'arbre. Bien qu'il soit possible de faire ceci avec des structures itératives, le moyen le plus facile est d'employer des sous-programmes récursifs afin de traiter :

- La racine
- Le sous-arbre gauche
- Le sous-arbre droit.

Ce type de parcours est dit **préfixé**. Le programme va d'abord traiter tous les éléments de gauche. Arrivé à une feuille, il remonte au nœud précédent, puis passe à droite pour traiter les éléments de gauche de celui-ci, puis remonte et ainsi de suite.

Dans l'arbre binaire de l'exemple, l'ordre de sortie est le suivant :

- Branche gauche : 12 -> 9 -> 8 (c'est une feuille)
- On remonte au nœud 9

- Branche droite : 10 (c'est une feuille)
- On remonte au nœud 9, puis la racine 12
- Branche droite : 14
- Branche gauche : 13 (feuille)
- On remonte au nœud 14
- Branche gauche : 16
- La sortie finale est donc : 12 9 8 10 14 13 16.

Pour le représenter, il faut utiliser une fonction ou procédure récursive.

```
Fonction prefixe(pNoeud :pointeur sur noeud)
Début
  Si pNoeud<>NIL Alors
    Afficher pNoeud→valeur // racine
    prefixe(pNoeud→pGauche) // sous-arbre gauche
    prefixe(pNoeud→pDroite) // sous-arbre droit
  FinSi
Fin
```

Il existe deux autres types de parcours. Le parcours **postfixé** qui traite dans cet ordre :

- Le sous-arbre gauche
- Le sous-arbre droit
- La racine

L'ordre de sortie est 8 10 9 13 16 14 12.

```
Fonction postfixe(pNoeud :pointeur sur noeud)
Début
  Si pNoeud<>NIL Alors
    prefixe(pNoeud→pGauche) // sous-arbre gauche
    prefixe(pNoeud→pDroite) // sous-arbre droit
    Afficher pNoeud→valeur // racine
  FinSi
Fin
```

Et le parcours **infixé**, appelé aussi parcours **symétrique** ou hiérarchique canonique. Ce parcours sera très utile par la suite. L'ordre est le suivant :

- Le sous-arbre gauche
- La racine
- Le sous-arbre droit.

Cette fois l'ordre de sortie est le suivant :

- Sous-arbre gauche : 8 9 10
- Racine : 12
- Sous-arbre droit : 13 14 16.

Vous obtenez la séquence 8 9 10 12 13 14 16. C'est très intéressant : l'arbre binaire donné en exemple n'a pas été choisi au hasard. Il s'agit d'un arbre binaire ordonné, construit de sorte qu'avec un parcours infixé les valeurs des différents nœuds sont triées.

```
Fonction infixe(pNoeud :pointeur sur noeud)
Début
  Si pNoeud<>NIL Alors
    infixe(pNoeud→pGauche) // sous-arbre gauche
    Afficher pNoeud→valeur // racine
    infixe(pNoeud→pDroite) // sous-arbre droit
FinSi
Fin
```

4. Arbre binaire ordonné

a. Principe

Un arbre binaire est ordonné si pour une valeur d'un nœud donné, la valeur du fils de gauche lui est inférieure et la valeur du fils de droite lui est supérieure.

pGauche→valeur < pEncours→valeur < pDroite→valeur

Imaginez que vous voulez ajouter la valeur 15 dans l'arbre :

- Comparez 15 à la racine 12 : c'est supérieur, direction le nœud de droite.
- Comparez 15 au nœud 14 : c'est supérieur, direction le nœud de droite.
- Comparez 15 au nœud 16 : c'est inférieur, direction le nœud de gauche.
- Il n'y a pas de nœud à gauche : placez 15 dans ce nouveau nœud.

Tous les parcours sont possibles et le parcours infixé vous donne toutes les valeurs déjà triées.

b. Recherche d'un élément

Pour rechercher un élément, vous avez deux solutions : utiliser une solution itérative ou une solution récursive. En effet les deux sont possibles et assez simples. Il suffit de comparer la valeur recherchée à la valeur de chaque nœud. Si elle est inférieure, alors la recherche continue à gauche, sinon elle continue à droite, tant qu'une feuille n'a pas été atteinte et que la valeur n'a pas été trouvée.

La fonction rech1() prend deux arguments : la valeur recherchée et la racine de l'arbre. Elle retourne un booléen VRAI si la valeur a été trouvée, FAUX sinon. Elle utilise une simple boucle.

```
Fonction rech1(vrech :entier, pArbre :pointeur sur noeud) :Booléen
Var
    trouve :booléen
    pEncours :pointeur sur noeud
Début
    pEncours=pArbre
    trouve=FAUX
    Tant que pEncours<>NIL ET trouve=FAUX Faire
      Si pEncours→valeur=vrech Alors
        trouve=VRAI
      Sinon
        Si vrech < pEncours→valeur Alors
          pEncours=pEncours→pGauche
        Sinon
          pEncours=pEncours→pDroite
        Finsi
      FinSi
    FinTantQue
Finfonc
```

La fonction rech2() est récursive. Elle prend trois arguments : la valeur recherchée, la racine de l'arbre et l'adresse du nœud contenant la valeur trouvée. Si la valeur n'est pas trouvée, l'adresse contient NIL.

```
Fonction rech2(vrech :entier, pArbre, pEncours,pointeurs sur noeud)
Début
  Si pArbre=NIL Alors
    pEncours=NIL
  Sinon
    Si pArbre→valeur=vrech Alors
      pEncours=pArbre
    Sinon
      Si pArbre→valeur > vrech Alors
        rech2(valeur,pArbre→pGauche, pEncours)
      Sinon
        rech2(valeur,pArbre→pDroite, pEncours)
      FinSi
    Finsi
  FinSi
FinFonc
```

c. Ajout d'un élément

Quand vous ajoutez un élément, vous devez respecter la structure de l'arbre ordonné. L'ajout d'un élément ajoute une feuille à l'arbre. Il vous faut trouver le chemin jusqu'au nœud père. La fonction rech2() peut être modifiée en ce sens : si l'élément à ajouter n'est pas trouvé, alors le dernier élément qui vaut alors NIL doit être remplacé par la nouvelle feuille et être raccordé à la bonne branche au père. Il faut donc conserver l'adresse du père.

La fonction inserer() prend trois arguments : la valeur à ajouter, l'adresse de l'arbre et l'adresse du nœud auquel ajouter la valeur, à droite ou à gauche selon le cas.

```
Fonction inserer(v:entier, pArbre, pPrec : pointeurs sur noeud)
Var
    pNouveau=pointeur sur noeud
Début
    Si pArbre=NIL Alors
      pNouveau=nouveau noeud
      pNouveau→valeur=v
      pNouveau→pGauche=NIL
      pNouveau→pDroite=NIL
      Si pPrec<> NIL Alors
        Si v>pPrec→Valeur Alors
             pPrec→pDroite=pNouveau
        Sinon
             pPrec→pGauche=pNouveau
        FinSI
      FinSi
    Sinon
      Si pArbre→valeur<>v Alors
        Si  v > pArbre→valeur Alors
           insérer (v, pArbre, pArbre→pDroite)
        Sinon
           insérer (v,pArbre, pArbre→pGauche)
        FinSi
      FinSi
    FinSi
FinFonc
```

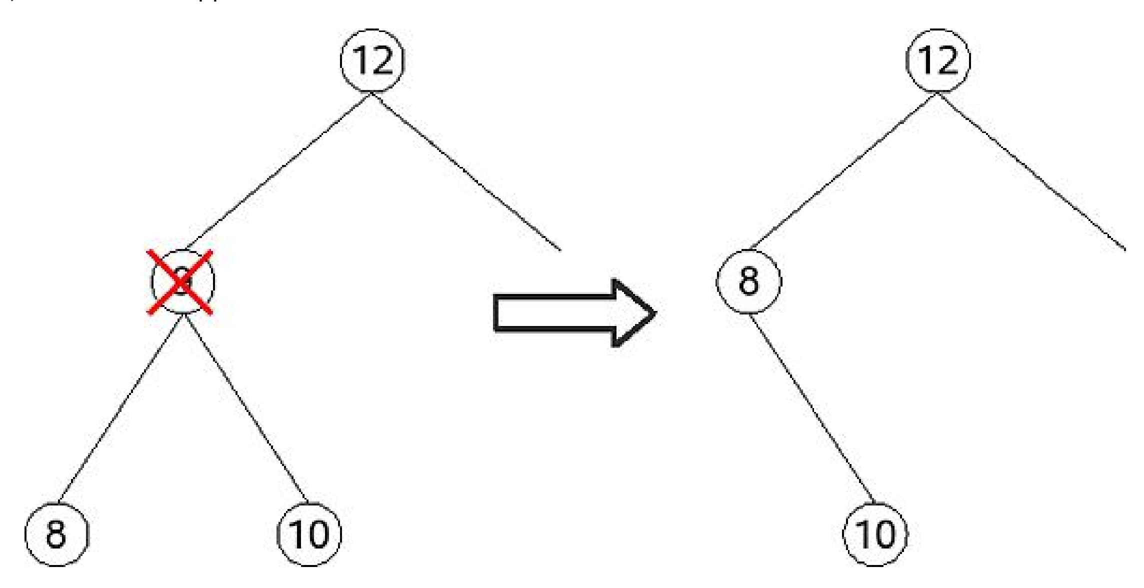
d. Suppression d'un nœud

Pour le dernier point de ce chapitre, c'est vous qui allez écrire l'algorithme de la fonction nécessaire à la suppression du nœud. Il y a trois cas à traiter :

- La suppression d'un nœud sans fils (une feuille), c'est le cas le plus simple. Le pointeur correspondant (droite ou gauche) du père doit être placé à NIL.

- La suppression d'un nœud ayant un fils : le fils droit être raccordé au bon pointeur du grand-père.
- La suppression d'un nœud ayant deux fils. C'est le cas le plus problématique.

Pour ce dernier cas, vous pouvez procéder aux chaînages directement. Par exemple, soit un morceau de l'arbre d'exemple, vous voulez supprimer le nœud 8 :



Suppression d'un nœud à deux fils

Le nœud 8 étant supprimé, il faut réorganiser l'arbre en conséquence. Le nœud avait deux fils : celui de gauche (et tout son sous-arbre) remplace le nœud supprimé, celui de droite (et tout son sous-arbre) va à droite du nouveau nœud.

Une autre possibilité est de supprimer le nœud, parcourir les deux sous-arbres de cet ancien nœud et d'ajouter chaque élément dans le premier arbre.

Principe de l'objet, une notion évidente

1. Avant de continuer

Vous voici dans le dernier chapitre de ce livre. Si votre objectif était d'apprendre à programmer dans des langages procéduraux ou fonctionnels, c'est-à-dire basés sur l'utilisation de sous-programmes tels que présentés ici, vous pourriez vous arrêter là. En effet, dès à présent vous disposez de tous les éléments indispensables pour programmer en langage C ou en Pascal.

Cependant, il serait dommage de ne pas continuer, ceci même si vous n'aurez pas tout de suite à programmer en objet. La programmation objet est depuis le début des années 1990 non seulement un classique, mais fait partie de la culture informatique. Un langage comme le C++, évolution du langage C et même d'autres comme Delphi dérivé du Pascal, Visual Basic dérivé du Basic, ainsi que la plupart des langages de macros sous MS Office ou OpenOffice.org, ou encore les langages de certains gestionnaires de bases de données, sont des langages objet.

Ne pas comprendre l'objet, c'est risquer de se couper de beaucoup de produits, beaucoup de fonctionnalités et parfois aussi d'une plus grande simplicité dans certains traitements.

2. Rappels sur la programmation procédurale

Les procédures ou fonctions reçoivent des données en arguments (paramètres). Elles retournent des valeurs soit par le même chemin (pointeurs ou référence), soit en renvoyant directement une donnée comme résultat de la fonction.

Dans un langage procédural (ou fonctionnel) les données sont séparées des programmes qui les utilisent.

a. Les données

Chaque variable dispose d'un type qui indique quelle sorte de valeur elle peut contenir. Ces types sont dits primitifs quand ils sont directement proposés par le langage lui-même. Ils peuvent différer selon les langages mais le C propose une série d'entiers, de réels et de caractères. D'autres incluent les chaînes de caractères.

Le type de la variable peut aussi être défini par le programmeur. Ce sont les types structurés que vous décrivez vous-même.

Les tableaux permettent de regrouper en un tout plusieurs occurrences de valeurs dans une même variable. Ils peuvent contenir n valeurs, généralement du même type (dans les langages typés comme le C) ou non (langages non typés comme le PHP).

Des variables particulières appelées les pointeurs ne contiennent pas directement une valeur mais l'adresse d'une variable contenant cette valeur, c'est-à-dire l'adresse de l'emplacement de cette valeur dans la mémoire de l'ordinateur. Bien que plus difficile à appréhender, elles permettent une souplesse inégalée dans les manipulations de valeurs. Pour pallier au risque de complexité dans certains langages évolués, la notion de référence remplace parfois celle de pointeur, notamment en Java ou en PHP.

b. Les traitements

Les traitements sont effectués soit dans le programme principal (qu'on appelle parfois le corps de programme) soit dans des sous-programmes appelés procédures ou fonctions. Dans ce dernier cas, la fonction est utilisée comme une instruction retournant une valeur, alors que la procédure est un bloc d'instructions sans valeur mais pouvant éventuellement en retourner via un passage de valeurs au travers des paramètres. Des langages comme le C, PHP ou Java ne font pas de distinction entre ces deux notions, une fonction ne retournant pas forcément de valeur.

3. L'objet

a. Dans la vie courante

Regardez autour de vous de quoi sont constituées tous les objets réels ou abstraits qui vous entourent. Un objet réel est par exemple une salière, un couteau, un stylo, une voiture, votre écran d'ordinateur, un téléphone, etc. Par abstrait comprenez une entreprise, un service, une organisation quelconque, etc.

Ces objets ont des propriétés intrinsèques. Prenez un écran, parmi ses propriétés physiques et d'usage vous trouvez :

- ses dimensions,
- son type (crt, lcd, etc),
- ses connecteurs (vga, dvi),
- son poids (c'est lourd un écran à tube),
- la diagonale d'affichage en pouces,
- les résolutions d'affichage supportées avec les fréquences associées,
- etc.

Ce même écran a probablement un mode d'emploi décrivant les manipulations tant matérielles que logicielles pour le paramétrer, par exemple :

- régler les couleurs (et associé : contraste, etc),
- régler la zone d'affichage,
- changer de résolution,
- etc.

Un objet comme un écran dispose donc de propriétés : c'est la description de ce qu'il est, de son état.

Il dispose aussi de méthodes : quelles sont les actions possibles pour modifier son état, son comportement, pour l'utiliser. C'est ce qu'il sait faire.

Tous les objets de la vie courante ont des propriétés et des méthodes. Même une salière (dimensions, couleur, nombre de trous, contenu, méthodes pour faire couler le sel plus ou moins vite, l'ouvrir, la remplir, la fermer, etc). Imaginez le nombre de propriétés et de méthodes pour un être humain…

La définition des propriétés et des méthodes décrites pour un écran est valable pour 99% des écrans si ce n'est plus. Le contenu des propriétés sera probablement modifié, mais si l'écran est standard le changement de résolution via Windows, MacOS ou Linux sera effectué de la même manière. La définition globale (propriétés et méthodes) d'un tel objet peut former une sorte de moule commun à tous les objets, les écrans, de même type.

Une fois que vous disposez de ce "moule", vous pouvez l'appliquer à autant d'objets, les écrans, que vous voulez, avec d'éventuelles variantes, en appliquant les mêmes méthodes.

b. En informatique

Programmation procédurale

En programmation procédurale, la question à se poser quand on développe est "À quoi sert le programme ?". Ici, il sert à manipuler les informations et les fonctions d'un écran.

Comment en programmation représenter les propriétés de l'écran et les manipulations qui y sont associées ? Jusqu'à présent vous auriez probablement raisonné ainsi :

- Regrouper toutes les informations sur l'écran dans un type structuré tEcran (par exemple).
- Créer des sous-programmes de gestion des paramètres de l'écran, prenant comme arguments la structure correspondante.
- Pour n écrans, vous créez n enregistrements (structures) de type tEcran, un par écran, dans des variables différentes, ou dans des tableaux.

Pour résumer, en algorithmique vous écririez quelque chose ressemblant plus ou moins à cela :

```
Type
  Structure tEcran
    type :chaîne
    marque :chaîne
    modèle :chaîne
    diagonale :entier
    hauteur :reel
    largeur :reel
    profondeur :reel
    poids :reel
    connecteur :chaîne
    resolution : tableau[1..10] de chaînes
  FinStruct
Procédure changer_resolution(...)
Procédure mise_en_veille(...)
Procédure rallumer(...)
Procédure regler_affichage(...)
...
```

C'est correct, possible, vous devrez probablement jouer avec les pointeurs et références pour le passage de la structure, donc pourquoi pas. Des milliers de programmes ont été développés ainsi. Posez-vous cependant la question : pourquoi ne pas associer les propriétés d'un objet et les traitements associés en un même ensemble ?

Programmation objet

Quand vous programmez en objet, la question à se poser est "Sur quoi porte le programme ?". La réponse est : sur les écrans. Puisque le programme porte sur les écrans, pourquoi ne pas tenter de répondre à la question posée à la fin du point précédent ? La réponse est "pourquoi pas" et c'est le but ou l'un des buts de la programmation objet.

En langage procédural ou en algorithmique, une structure est déjà appelée un objet, dans un certain nombre d'ouvrages d'algorithmique un peu anciens (années 1980), dans le sens où elle contient un ensemble de données cohérentes sur un sujet bien précis. La structure tarticle contenait toutes les propriétés d'un article. Vous accédez à une information à travers le point "." ou la flèche "→" si l'enregistrement est une référence.

La structure code donc toutes les propriétés d'un objet réel ou abstrait. Cependant tous les traitements associés sont dans des sous-programmes à part.

Regardez ces lignes issues de l'exemple en PHP sur les structures :

```
  class tarticle {
    public $ref;
    public $libelle;
    public $prix;
    public $fab;
  }


  $article=new tarticle();
  $article->ref="Art001_01";
```

La variable $article est en principe une structure. Pourtant, ce n'est pas le cas. C'est une variable objet (une instance). Vous accédez à une propriété, une donnée de l'objet avec la flèche "->". Vous accédez non pas à des enregistrements mais à des attributs associés à la variable. Voici une définition de l'objet :

Dans les langages objets, les données et les traitements qui manipulent ces données sont regroupés au sein d'une même entité appelée objet.

En PHP les objets sont décrits avec le mot-clé "**class**". En algorithmique ils le sont avec le mot-clé "**classe**".

Quelles sont les propriétés d'une chaîne de caractères ?

- La chaîne elle-même, composée d'une suite de caractères.

Quels sont les traitements applicables à une chaîne ?

- Le calcul de sa longueur,
- le découpage en sous-chaînes,
- la concaténation avec une autre chaîne,

- la recherche d'une autre chaîne dedans,
- l'éclatement de la chaîne selon un délimiteur,
- la conversion en minuscules ou majuscules,
- la suppression des espaces avant et après,
- etc.

Maintenant, regroupez les propriétés de la chaîne et tous les traitements possibles dessus en un seul tout. Vous obtenez un objet.

Un programme objet est constitué d'un ensemble d'objets qui communiquent entre eux par l'émission et la réception de messages pour réaliser le traitement final. Cela paraît impressionnant et compliqué comme cela mais c'est en fait très simple. Dans la ligne suivante :

```
$cpt=$tab.count();
```

l'objet appelé $tab, un tableau PHP déclaré comme arrayObject, reçoit l'ordre associé à sa fonction count() : compter le nombre d'éléments de ce tableau.

Si vous n'avez pas encore bien compris, alors dites-vous qu'un véritable objet en informatique est un peu comme un enregistrement (il en reprend les mêmes propriétés que sont les champs) auquel on aurait ajouté dedans des sous-programmes chargés de manipuler ses champs. Les sous-programmes ne seraient plus à part mais partie prenante de la structure, qui serait alors non plus la définition d'un enregistrement, mais d'un objet.

4. Classe, objets

Il est nécessaire de connaître quelques mots de vocabulaire pour définir un objet. Un objet est défini par :

- Les données sur lui-même : ses propriétés, son état.
- Ce qu'il fait : les traitements qu'il associe à ses données ou les données reçues des autres objets.

Comme les structures se définissent avec le mot-clé du même nom, la structure d'un objet se décrit avec le mot-clé "**Classe**" pour classe d'objet. Cette structure de l'objet définit ce que sera un objet de ce type. La classe est le type de l'objet.

- Un objet est une variable dont le type est sa classe.
- Une classe est en fait un moule servant à créer plusieurs objets. Les objets issus d'une même classe sont différents les uns des autres parce que les valeurs de leurs propriétés ne sont pas toujours les mêmes.
- Comme il peut y avoir plusieurs objets de même type de classe, on dit qu'un objet est une **instance de classe**.

La définition d'un objet est un peu comme celle d'une structure, sauf qu'elle est composée de deux parties :

- Les **attributs** sont les propriétés de l'objet, c'est-à-dire les différentes variables qui définissent ce qu'il représente, son état.
- Les **méthodes** sont les sous-programmes, fonctions ou procédures qui influent sur l'objet, par exemple sur les attributs.

Ces deux parties sont appelées les **membres** de l'objet.

Les attributs sont des variables de n'importe quel type, primitif, tableau, objet, etc. Les méthodes sont l'équivalent des sous-programmes vus jusqu'à présent, mais propres à l'objet.

```
Classe mon_objet
  attributs
    attr1 :entier
    attr2 :tableau[1..10] de réels
    ...
  méthodes
    procédure afficher()
    procédure effacer()
  ...
FinClasse
```

Reprenez l'exemple procédural sur l'écran. Voici ce que pourrait être la classe de l'objet de type Ecran :

```
Type
Classe Ecran
  attributs
    type :Chaîne
    marque :chaîne
    modèle :chaîne
    diagonale :entier
    hauteur :reel
    largeur :reel
    profondeur :reel
    poids :reel
    connecteur :chaîne
    resolution : tableau[1..10] de chaînes
  méthodes
    Procédure changer_resolution(...)
    Procédure mise_en_veille(...)
    Procédure rallumer(...)
    Procédure regler_affichage(...)
    Procédure affiche_modèle()
    Procédure saisie_modèle()
FinClasse
```

5. Déclaration et accès

Un objet se déclare comme une variable ou un enregistrement sauf que le nom du type est le nom de la classe :

```
Var
  ecran1 :Ecran
  ecran2 :Ecran
```

Vous pouvez aussi créer des tableaux d'objets :

```
Var
  ecrans :tableau[1..10] d'Ecrans
```

Et même déclarer des pointeurs sur des objets, puisque eux aussi occupent une zone mémoire :

```
Var
  ecran :Ecran
  pEcran :pointeur sur Ecran
```

Par convention, les variables objets, qui seront simplement appelées objets par la suite, commencent souvent par la lettre o : oEcran1, o1, o2, etc. Mais vous n'êtes absolument pas obligé de suivre cette convention, sauf si elle vous est imposée.

Vous accédez aux divers membres de l'objet en utilisant le point "." entre le nom de l'objet et ses membres, exactement comme pour une structure. La seule différence est qu'après le point vous n'avez plus forcément un champ (un attribut) mais aussi un sous-programme (une méthode). Quand vous placez une méthode après le point, vous lancez l'exécution de celle-ci.

```
Type
  Classe Ecran
  ...
```

```
  FinClasse
Programme obj1
Var
  o1 :Ecran
Début
  // modification des attributs
  o1.type←"LCD"
  o1.connecteur←"VGA"

  // accès direct aux attributs
  Afficher o1.type

  // appel aux méthodes
  o1.mise_en_veille()
  o1.ralumer()
  o1.affiche_modèle()
Fin
```

6. Les méthodes

Regrouper les fonctions et les données au sein d'une même structure appelée objet est déjà bien plus cohérent pour la pensée. Mais comment manipuler les attributs de l'objet au sein d'une de ses méthodes ? C'est là qu'est un très gros avantage : alors qu'en programmation procédurale vous deviez passer la donnée (la variable) en argument, là c'est totalement inutile : la méthode "sait" implicitement que l'attribut auquel il accède appartient à l'objet :

```
Classe Ecran
  attributs
    type :chaîne
    ...
  méthodes
    Procédure modifie_type(E :t :chaîne)
    Début
      type←t
    FinProc
    ...
FinClasse
```

Il arrive parfois que des noms d'attributs soient identiques à ceux d'autres objets ou arguments passés en paramètres de méthodes. Pour éviter la confusion, vous pouvez explicitement désigner que le membre appartient à votre objet en précisant le mot-clé "**this**" suivi du point et de son nom. This sera remplacé par l'objet lui-même lors de l'appel.

```
Classe Ecran
    ...
    Procédure modifie_type(E :t :chaîne)
    Début
      this.type←t
    FinProc
    ...
FinClasse
```

En programmation objet pure, il n'y a plus de notion de programme principal et de sous-programmes : tous les traitements se font au sein des méthodes et même le bloc d'instructions principal est une méthode au sein d'une classe, comme en Java.

Comme tous les composants d'un programme sont des objets, il s'établit une communication entre les objets. En pratique, c'est évidemment vous qui initiez cette communication : quand on dit qu'un premier objet demande quelque chose à un second, c'est que l'une des méthodes du premier objet appelle l'une des méthodes du second. Puisque tout est objet, vous pouvez créer une instance du second objet comme attribut du premier, ou même passer des objets en arguments de méthodes.

Dans l'exemple précédent, l'objet de type Ecran peut recevoir le message `modifie_type` qui modifie l'attribut type.

Il est possible de décrire (programmer) une méthode en dehors de la définition de la classe (en algorithmique). Dans ce cas, dans la définition de la classe vous écrivez le prototype de la méthode (nom+paramètres) et en dessous vous définissez la méthode. Vous devez respecter la syntaxe suivante avec les doubles points "**::**" entre le nom de la classe et la méthode :

```
Procédure classe::methode(params)
```

Par exemple :

```
Classe Ecran
  ...
  méthodes
    Procédure modifie_type(E :t :chaîne)
    ...
FinClasse

Procédure Ecran::modifie_type(E :t :chaîne)
Début
  this.type←t
FinProc
```

7. Portée des membres

Dans le programme obj1, l'accès aux divers membres, attributs ou méthodes, se fait directement. Le programme accède aux membres en passant par l'objet et l'opérateur point. C'est que dans la définition de la classe, les membres sont bien souvent publics : ils sont accessibles sans aucune protection.

Vous pouvez choisir de rendre les divers membres publics ou privés. Par défaut, en PHP par exemple, si vous ne précisez rien tous sont publics. En algorithmique les attributs sont par défaut privés et les méthodes publiques.

- **Public** : les membres sont directement accessibles depuis toute autre partie du programme ou objet comme dans les exemples précédents via l'opérateur point.

- **Privé** : les membres ne sont plus accessibles depuis l'extérieur de l'objet. Ils sont seulement accessibles depuis l'intérieur de l'objet.

Vous pouvez préciser dans les déclarations le type d'accès à vos membres. Voici un exemple de la classe Ecran où les attributs sont privés, donc accessibles uniquement depuis les méthodes de l'objet et les méthodes publiques :

```
Classe Ecran
  attributs privés
    type :chaîne
    marque :chaîne
    modèle :chaîne
    ...
  méthodes publiques
    Procédure regler_affichage(...)
    Procédure affiche_modèle()
    Procédure saisie_modèle()
FinClasse
```

Avec cette nouvelle définition, vous ne pouvez plus utiliser le programme obj1 tel quel. Vous devez le modifier.

```
Programme obj2
Var
  o1:Ecran
Début
  // Attributs privés : accès depuis l'extérieur interdit
  o1.type←"LCD" // *** INTERDIT !!! ***
  // Accès direct aux attributs
  Afficher o1.type // *** INTERDIT !!! ***

  // Méthodes publiques : accès autorisé
  o1.saisie_modèle("Multisync FE1250+")o1.affiche_modèle()
Fin
```

Il existe un troisième type d'accès : l'accès **protégé**. Il est particulier et vous le reverrez plus loin lorsque la notion d'héritage sera abordée. Un membre protégé est considéré comme privé pour les autres objets indépendants car il n'est pas accessible depuis ceux-ci, mais comme public au sein de l'objet et des objets qui en dérivent.

8. Encapsulation des données

Les membres peuvent être publics ou privés. La structure interne d'un objet comporte un certain nombre d'attributs et de méthodes qui lui sont propres, qui reflètent sa structure interne. Des attributs et méthodes de l'objet ne sont utiles que pour d'autres méthodes et donc ne doivent pas être accessibles depuis l'extérieur de l'objet. De même, afin de laisser un seul point d'entrée aux divers attributs, vous ne devriez pas permettre l'accès à ceux-ci à la fois directement et par des méthodes.

Une bonne pratique consiste à interdire l'accès public aux attributs. Seules les méthodes y accèdent et si vous voulez les manipuler, vous définissez des méthodes publiques pour cela. Vous évitez ainsi que le programmeur manipule directement la structure interne de votre objet et y mette n'importe quoi car vous pouvez effectuer un contrôle au sein des méthodes. C'est le principe de l'**encapsulation** des données :

- Les attributs sont privés (ou protégés).
- Vous les manipulez en passant par des méthodes publiques.

L'exemple modifié suivant met en évidence ce qui aurait pu poser un problème : vous voulez modifier le type d'écran. En passant directement par un attribut public vous auriez pu mettre n'importe quoi dedans comme "PLAT".

Or la méthode modifie_type vérifie avant ce que vous y mettez dans un choix prédéfini (crt, lcd, plasma). Si le type que vous voulez ne correspond pas, il est rejeté. Vous venez de blinder (un peu) votre objet en empêchant quiconque de le "casser". Vous reconnaissez maintenant l'importance, dans certains cas, de l'encapsulation des données.

```
Types
Classe Ecran
  attributs privés
    ...
    type:chaîne
    ...
  méthodes publiques
    ...
    Fonction modif_type(mod :chaîne) :booléen
    ...
FinClasse
Fonction Ecran::modif_type(mod :chaîne) :booléen
Var
  tmod :tableau[1..3]<-{"CRT","LCD","PLASMA"} de chaînes
  ok :booléen
  i :entier
Début
  ok←FAUX
  i←1
  Tant que i<=3 ET NON ok Faire
    Si mod=tmod[i] Alors
      ok←VRAI
    FinSi
    i=i+1
  FinTantQue
  Si ok Alors
    this.type←mod
  FinSi
  Retourne OK
FinFonc
Programme obj3
Var
  oEcran :Ecran

Début
  Si NON oEcran.modifie_type("PLAT") Alors
    Afficher "Erreur dans la modification du type"
  FinSi
Fin
```

Les utilisateurs de vos classes n'ont pas toujours à savoir comment sont définies celles-ci : ils n'ont accès qu'aux membres publics de votre classe. Vous définissez en fait des **interfaces** (la liste des méthodes) depuis lesquelles les utilisateurs accèdent indirectement aux attributs. Les méthodes qui permettent de manipuler l'objet sont les mêmes quelles que soient les valeurs des attributs de l'objet. C'est pourquoi un objet dont l'attribut type est CRT ou LCD

s'utilise de la même manière : c'est le principe d'**abstraction** : l'utilisateur manipule n'importe quelle instance de classe sans rien connaître de sa complexité interne.

9. L'héritage

a. Principe

Vous ne trouverez la notion d'héritage nulle part ailleurs que dans l'objet. Il permet de créer une nouvelle classe depuis une classe existante. On dit alors que la nouvelle classe **hérite** de la première, ou qu'elle **dérive** de la première. Vous entendrez régulièrement parler de classes dérivées en programmation objet.

Quand une classe hérite d'une autre, elle récupère tous ses attributs et méthodes. On dit que la classe de base est une **superclasse**. La classe dérivée hérite des membres de la superclasse. Sauf que dans la classe dérivée, vous pouvez ajouter des nouveaux attributs et des nouvelles méthodes et même redéfinir les méthodes de la classe de base : c'est une fonctionnalité majeure de la notion d'objet.

Soit une classe animal :

```
Classe animal
  attributs privés
    forme :chaîne
    ordre :chaîne
    ...
  méthodes publiques
    Procédure modifie_ordre(E :ordre :chaîne)
    ...
FinClasse
```

Le règne animal est complexe et pourraît être divisé en énormément de choses : vertébrés, invertébrés, mammifères, insectes ou encore herbivores, carnivores ou omnivores. Ces trois derniers sont à la base tous des animaux : des classes de ce genre dérivent de la classe animal.

Pour déclarer une classe qui dérive d'une autre, respectez la syntaxe suivante :

```
Classe maClasse hérite de Superclasse
  attributs
    ...
  méthodes
FinClasse
```

Pour définir les classes herbivores, carnivores ou omnivores, vous héritez des propriétés de base du sous-règne animal associé.

```
Classe herbivore hérite de animal
  ...
FinClasse
Classe carnivore hérite de animal
  ...
FinClasse
Classe omnivore hérite de animal
  ...
FinClasse
```

Avec l'héritage, vous pouvez concevoir des classes de plus en plus spécialisées. De même si vous devez concevoir des classes dont les propriétés sont très proches, plutôt que de tout reprogrammer depuis zéro, vous pouvez concevoir une classe de base (la superclasse) et créer deux classes qui héritent de celle-ci.

Donnez un ou deux noms d'animaux dans chaque catégorie. La vache est herbivore, le cheval aussi. Les deux n'ont pas le même régime et pas le même nombre d'estomacs. Vous pouvez les différencier dans deux classes différentes qui dérivent de la classe herbivore :

```
Classe cheval hérite de herbivore
  ...
FinClasse
Classe vache hérite de herbivore
  ...
FinClasse
```

Vous pouvez faire de même avec les carnivores : lion, chien et les omnivores : l'homme, le singe. Vous constituez donc un ensemble de classes dérivées de plus en plus spécialisées.

Dans une classe dérivée, vous avez un accès direct aux membres, tant les attributs que les méthodes, de toutes les superclasses d'origine, en cascade. Donc ceci est correct, partant du principe que omnivore dérive de animal vous pouvez appeler la méthode modifie_ordre() de la superclasse animal.

```
Programme obj4
Var
  o3:herbivore
Début
  o3.modifie_ordre(mammifère)
Fin
```

Rappelez-vous cependant que l'accès aux attributs dépend de leur protection. Si les attributs de la superclasse sont privés, une classe dérivée ne peut pas y accéder directement. Si elle est protégée ou publique, c'est possible.

b. Commerce

L'avantage est de taille, à tel point qu'il existe pour des langages comme C++, PHP ou Java des éditeurs de logiciels spécialisés dans la vente de classes, regroupées sous forme de bibliothèques. Il existe donc un commerce d'objets et comme les classes sont des moules réutilisables à volonté tout le monde y trouve son compte. Vous trouverez ces produits sous le nom d'API (*Application Programming Interface*). Il existe des API pour à peu près tous les domaines de l'informatique : telle API pour accéder plus facilement aux bases de données, telle API pour aider au développement d'un logiciel de gestion de cabinet médical, etc. Vous trouverez sur Internet des sites spécialisés dans la diffusion, souvent gratuite, de classes très pratiques. Si la classe de base de PHP pour la gestion des tableaux ne vous suffit pas, vous pourrez en trouver d'autres qui proposeront tous les types de tri, de reconstituer une chaîne, d'allouer dynamiquement des éléments supplémentaires, qui gèreront les listes chaînées à votre place, etc. Tout est possible.

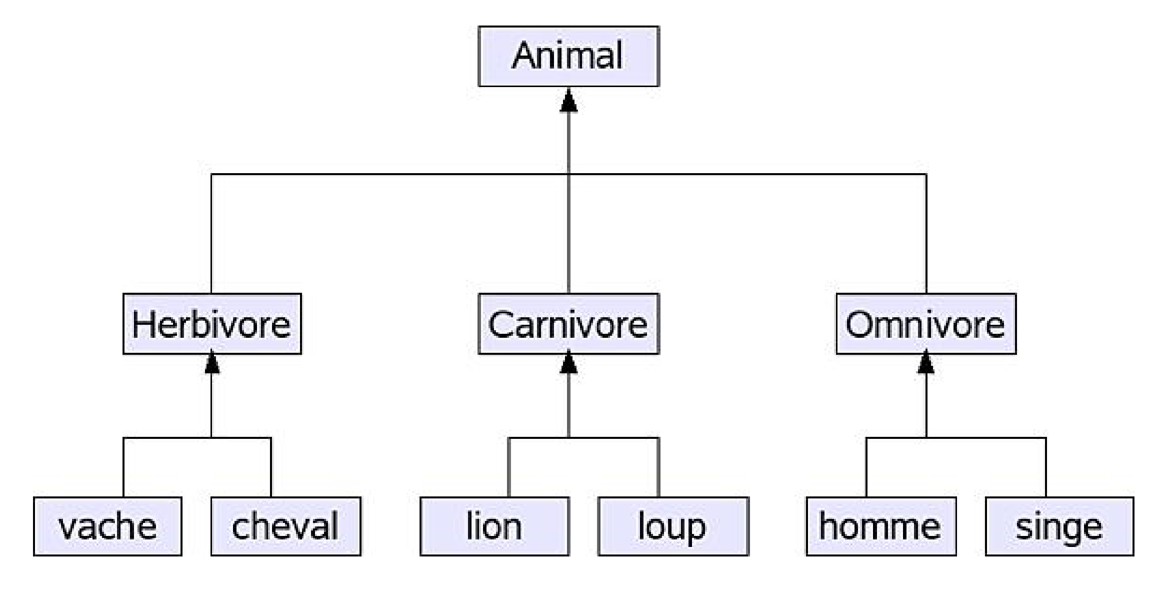
En une phrase, la grande force de l'objet est d'être réutilisable, accélérant ainsi le développement d'applications par l'utilisation de composants déjà testés et validés.

c. Hiérarchie

Quand une classe hérite d'une autre, on dit que c'est une classe fille (et l'objet résultat un fils). Il y a donc une relation hiérarchique reprise sur la généalogie. Il y a des objets père, grand-père, etc, jusqu'à une classe de base. Comme plusieurs classes distinctes peuvent dériver d'une superclasse et que d'autres encore peuvent dériver en cascade des nouvelles classes, vous obtenez un hiérarchie entre les classes, un arbre des classes (et objets associés), comme un arbre généalogique : une classe de base a tant de fils, qui ont eux-mêmes tant de fils et ainsi de suite. Cette hiérarchie décrit une arborescence.

C'est flagrant en Java où toutes les classes héritent à la base d'une seule superclasse appelée **Object**. Cette classe sert de prototype, de base, pour toutes les autres. Toutes les classes proposées par défaut par Java (toutes les API du SDK) dérivent de la classe Object. La classe Object propose des méthodes qui sont toutes réimplémentées dans les classes dérivées. La classe Object est donc la superclasse de toutes les classes Java.

Si vous reprenez l'exemple des régimes alimentaires du règne animal, vous obtenez une arborescence de ce genre :

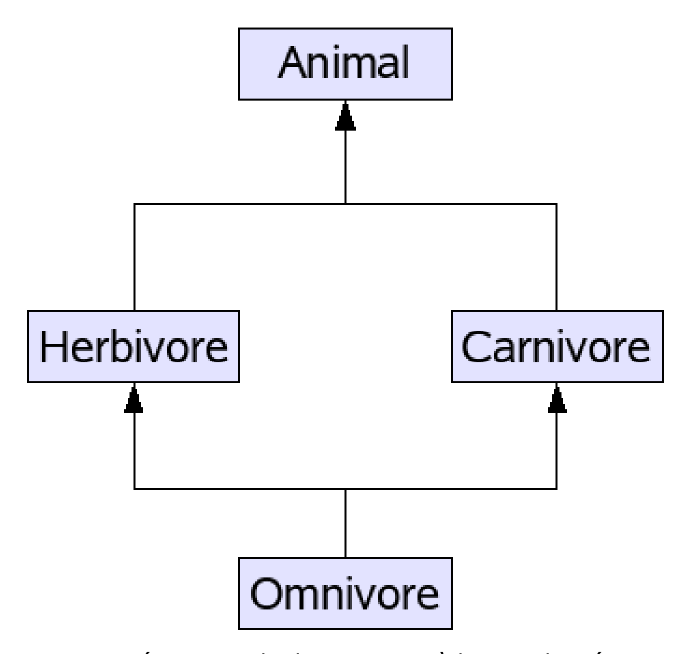


d. Simple ou multiple

N classes peuvent dériver d'une superclasse, c'est ce que vous avez vu jusqu'à présent. Si vous regardez la hiérarchie actuelle, il serait éventuellement possible de modifier quelque chose. Un omnivore mange à la fois de l'herbe (fruits et légumes) et de la viande. Il est donc à la fois Herbivore et carnivore. Pourrait-on alors créer une classe Omnivore dérivant des classes Carnivore et Herbivore ? C'est possible. Dans ce cas, la nouvelle classe hérite des membres des deux classes pères.

```
Classe omnivore hérite de Herbivore, Carnivore
   ...
FinClasse
```

La hiérarchie devient :



Héritage multiple, attention à la complexité

C'est possible, mais cela pose souvent plus de problèmes que cela n'en résout. Les avis sont partagés sur ce sujet. Mais voici un bref exemple des implications un peu tordues générées par l'héritage multiple : si deux méthodes de même nom sont définies dans les deux superclasses et pas dans la classe dérivée, quelle méthode doit être appelée si son nom est appelé dans celle-ci ? Les langages proposent des astuces pour gérer ce genre de cas, mais cela devient très complexe. L'héritage multiple a été implémenté dans le langage C++ mais pas en Java ou PHP, tout au moins pas directement : une classe ne dérive que d'une seule classe.

Avant de vous lancer dans de l'héritage multiple, tentez de voir s'il est possible de transformer la hiérarchie en héritage simple, d'utiliser des classes instanciant comme attributs d'autres classes ou d'utiliser les interfaces. Il n'y a quasiment jamais d'obligation à utiliser l'héritage multiple.

10. Le polymorphisme

a. Principe

Quand vous héritez d'une classe, vous pouvez depuis un objet de la nouvelle classe accéder directement aux méthodes des superclasses : de fait elles font toutes partie de la classe dérivée. Pourtant il est probable que si vous avez dérivé de la superclasse c'est que vous avez trouvé un avantage à ajouter des nouveaux attributs et des nouvelles méthodes. Certaines de ces nouvelles méthodes remplacent peut-être les méthodes de la superclasse ou y font appel.

Le polymorphisme des méthodes est un autre concept essentiel de la programmation objet. Polymorphisme signifie qu'une même chose peut prendre plusieurs formes. Certains animaux en font leur spécialité comme les abeilles : chez une même espèce, il y a trois formes d'individus, la femelle (reine), les mâles (faux-bourdon) et les ouvrières (femelles stériles). Trois formes différentes pour la même espèce, ou plutôt la même larve à la base.

Du côté de l’objet, le polymorphisme permet de créer plusieurs méthodes qui portent le même nom. On distingue trois types de polymorphisme.

b. Le polymorphisme ad hoc

Généralement dans un langage procédural, vous ne pouvez avoir qu’une seule procédure portant le même nom dans un même programme. Si le sous-programme Afficher() doit être utilisé pour des données différentes, vous devez soit l’adapter en conséquence, soit créer autant de sous-programme Afficher_xxx() où xxx nommerait le rôle exact du sous-programme.

En objet, comme chaque définition de classe est indépendante de sa voisine, vous pouvez réutiliser les mêmes noms de méthodes dans des classes différentes, puisqu’elles n’ont aucun rapport entre elles.

```
Classe lion hérite de Carnivore
  ...
  méthodes publiques
    Procédure affiche_nom()
    ...
FinClasse
Classe Ecran
  ...
  méthodes publiques
    Procédure affiche_nom()
    ...
FinClasse
```

c. Le polymorphisme d’héritage

Vous avez le droit et c’est souvent le cas, de redéfinir une méthode d’une superclasse dans une classe dérivée, avec le même nom de méthode. Une classe Object comme en Java définit des méthodes de base, qui sont redéfinies dans les classes dérivées en fonction des nouvelles propriétés de celle-ci. Un exemple classique est un jeu d’échecs et ses pièces. Chaque pièce du jeu a un nombre plus ou moins limité de mouvements autorisés.

Une classe pièce va définir une méthode mouvement(). Les classes dérivées pion, fou, tour, cavalier, roi et reine vont contenir aussi une méthode mouvement() qui déterminera leurs mouvements propres :

```
Classe Pièce
  ...
  méthodes publiques
    Procédure mouvement()
    Début
      Afficher "Mouvement de Pièce"
    FinProc
  ...
FinClasse
Classe Tour hérite de Pièce
  ...
  méthodes publiques
    Procédure mouvement()
    Début
      Afficher "Mouvement de Tour"
    FinProc
    ...
FinClasse
```

Quand vous allez déclarer un objet de type tour et que vous allez accéder à la méthode mouvement(), c’est la méthode mouvement() de Pièce qui va être appelée et c’est donc le message correspondant "Mouvement de Tour" qui s’affiche.

```
Programme obj5
var
  matour1 :Tour
Début
  ...
  // Appel de mouvement() de l’objet matour1
  matour1.mouvement()
Fin
```

Vous pouvez explicitement faire appel à la méthode de la superclasse Pièce dont Tour dérive à l'aide du mot-clé **super** (pour superclasse) au sein de la ou des méthodes concernées. Dans l'exemple suivant la méthode mouvement() de la classe Tour a été modifiée pour appeler la méthode mouvement() de Pièce. À l'issue de ce programme, les deux messages sont affichés, d'abord celui pour Pièce, puis celui pour Tour.

```
Classe Tour hérite de Pièce
  ...
  méthodes publiques
    Procédure mouvement()
    Début
      // Appel de mouvement() de Pièce
      super.mouvement()
      Afficher "Mouvement de Tour"
    FinProc
    ...
FinClasse
Programme obj6
var
  matour1 :Tour
Début
  ...
  // Appel de mouvement() de l'objet matour1
  matour1.mouvement()
Fin
```

En cas d'héritage en cascade, le mot-clé super se rapporte à la classe mère, c'est-à-dire celle dont dérive directement la classe fille. Pour remonter toute la hiérarchie des classes, vous devez utiliser le mot-clé super dans toutes les méthodes de toutes les classes précédentes, en cascade.

d. Le polymorphisme paramétrique

Chaque méthode au sein d'une classe dispose d'une **signature** particulière. Cette signature est constituée du nom de la méthode, des paramètres qu'elle prend et de la valeur qu'elle retourne. Le langage objet se base sur cette signature pour appeler la bonne méthode.

Le principe de l'objet indique qu'il suffit qu'un seul de ces trois constituants de la signature varie pour que la méthode soit considérée comme différente. Vous pouvez faire par exemple varier le type de variable retournée par la méthode et aussi les paramètres, mais pas le nom de celle-ci ? Voyez-vous les implications directes ? Vous avez le droit de donner le même nom à plusieurs méthodes à condition que le nombre ou les types des paramètres de celles-ci ne soient pas identiques.

Soit une classe calcul qui redéfinit des opérations sur divers types de variables : entiers, réels, mais aussi chaînes de caractères pour les concaténer par exemple. Dans cette classe, vous voulez que quand vous appelez la méthode addition(), l'addition des deux valeurs passées en paramètres soit effectuée, sans se soucier de leur type. Cela ne représente aucun problème. L'exemple suivant implémente ces trois méthodes et le programme principal y fait appel.

```
 Classe calcul
  ...
  méthodes publiques
  // Addition de deux entiers
  Fonction addition(x,y :entiers) :entier
  Début
    Retourne x+y
  FinFonc

  // Addition de deux réels
  Fonction addition(x,y :réels) :réel
  Début
    Retourne x+y
  FinFonc

  // Concaténation de deux chaînes
  Fonction addition(x,y :chaînes) :chaîne
  Début
    Retourne x&y
```

```
  FinFonc
  ...
FinClasse
Programme obj7
Var
  oCalc :Calcul
Début
  // Deux entiers
  Afficher oCalc.addition(10,20)

  // Deux réels
  Affichier oCalc.addition(3.1415927,14.984)

  // Deux chaînes
  Afficher oCalc.addition("Bonjour" ,"les amis")
Fin
```

Manipuler les objets

1. Les constructeurs

a. Déclaration

Quand vous instanciez une classe, c'est-à-dire quand vous créez un objet du type de classe donné, il vous faut bien souvent appeler diverses méthodes pour remplir ses attributs avec les bonnes valeurs. Pour reprendre l'exemple de la classe Ecran, nul doute qu'en créant un objet de ce type vous voudrez positionner les attributs type, marque, modèle, etc, aux bonnes valeurs. La solution actuelle consiste à appeler explicitement les méthodes prévues à cet effet : modif_type(), modif_ marque(), modif_modèle() et ainsi de suite.

Il existe un moyen plus efficace et implicite de positionner les bonnes valeurs des attributs (et d'effectuer toute autre opération nécessaire) dès l'instanciation. Ce moyen, c'est le **constructeur**.

Le constructeur est une méthode particulière qui est appelée automatiquement dès que vous créez un objet sans que vous ayez à le préciser, que se soit par déclaration ou allocation dynamique (via un pointeur). Dans cette méthode, libre à vous de faire ce que vous voulez, mais dans 90% des cas son rôle sera de donner aux attributs leurs bonnes valeurs initiales, ou des valeurs par défaut.

Un constructeur doit respecter les deux propriétés suivantes :

- Il porte le même nom que la classe.
- Il ne retourne pas de valeur.

Via le polymorphisme paramétrique (qu'on appelle aussi la surcharge des méthodes) vous pouvez créer autant de constructeurs que vous le souhaitez, un par cas selon les paramètres passés. En algorithmique vous ajouterez le mot-clé Constructeur devant le nom de la méthode en remplacement de Procédure ou Fonction. Ce n'est pas forcément obligatoire car aucune autre méthode ne doit avoir le même nom que la classe.

b. Appel implicite

La classe Ecran se voit complétée de deux constructeurs : le premier ne reçoit aucun paramètre et sera utilisé lorsque l'objet ne reçoit aucune valeur lors de sa création. Le second prend trois paramètres (type, marque, modèle) pour les initialiser à la création de l'objet.



Un constructeur qui ne reçoit aucun paramètre est appelé le constructeur par défaut. Il est appelé quand aucune valeur n'est passée à l'objet lors de sa création.

```
Classe Ecran
  attributs privés
    type :chaîne
    marque :chaîne
    modèle :chaîne
  méthodes publiques
    Constructeur Ecran()
    Début
      Afficher "Constructeur par défaut"
      this.type←"Inconnu"
      this.marque←"Inconnu"
      this.modèle←"Inconnu"
    Fin
    Constructeur Ecran(t,mq,mdl :chaînes)
    Début
      Afficher "Constructeur avec 3 arguments"
      this.type←t
      this.marque←mq
      this.modèle←mdl
    Fin
FinClasse
```

Comment savoir quel constructeur va être appelé à la création de l'objet ? C'est à vous de passer les bonnes valeurs à l'objet dès son instanciation (déclaration ou création). Dans l'exemple suivant, deux objets Ecran sont créés. Le premier l'est comme d'habitude : c'est le constructeur par défaut (sans paramètres) qui sera appelé. Le second prend trois paramètres entre parenthèses, comme si vous passiez des paramètres à une méthode : c'est le constructeur associé qui sera appelé.

```
Programme obj8
Var
  // Appel implicite à Ecran()
  o1:Ecran
  // Appel implicite à Ecran(t,mq,mdl)
  o2:Ecran("LCD","LG","L1915S")
Début
  o1.affiche_modèle() // INCONNU
  o2.affiche_modèle() // L1915S
Fin
```

À l'exécution sans avoir rien demandé, les deux messages au sein des constructeurs sont affichés, les attributs disposent déjà de valeurs, placées là par les constructeurs.

Avec une allocation dynamique, c'est pareil :

```
Programme obj9
Var
  p_oEcran :pointeur sur Ecran
Début
  p_oEcran←nouveau Ecran("LCD","LG","L1915S")
  p_oEcran->affiche_modèle() // L1915S
  ...
Fin
```

c. L'héritage

Que se passe-t-il lorsque vous instanciez un objet d'une classe dérivée ?

- Si la classe dérivée n'a pas de constructeur, c'est celui de la superclasse qui est appelé, si c'est possible (paramètres identiques par exemple).
- Si la classe dérivée dispose d'un constructeur, c'est lui qui est appelé.
- Par défaut si le constructeur d'une classe dérivée est présent, le constructeur de la superclasse n'est pas appelé. C'est à vous d'y faire appel explicitement.

```
Classe A
  ...
  Constructeur A()
  Début
    Afficher "Je suis A"
  Fin
  ...
FinClasse

Classe B hérite de A
  ...
  Constructeur B()
  Début
    Afficher "Je suis B"
  Fin
  ...
FinClasse

Programme obj10
Var
  o1 :A
  o2 :B
Début
```

```
  ...
Fin
```

Le programme obj10 crée deux objets et donne l'affichage suivant :

```
Je suis A
Je suis B
```

Pour appeler explicitement le constructeur de A depuis B, vous devez le faire explicitement au sein du constructeur de B via le mot-clé **super**. Celui-ci remplaçant l'objet superclasse, il s'utilise comme un objet. Modifiez le constructeur de B ainsi :

```
Classe B hérite de A
  ...
  Constructeur B()
  Début
    super()
    Afficher "Je suis B"
  Fin
  ...
FinClasse
```

Vous obtenez en relançant obj10 le résultat suivant :

```
Je suis A
Je suis A
Je suis B
```

Une nouvelle ligne correspondant à l'appel du constructeur de A depuis B est affichée. Là encore, en cas d'héritage en cascade, vous devez appeler les constructeurs en cascade.

Il faut bien comprendre pourquoi l'appel au constructeur de la superclasse n'est pas automatique : rien ne vous empêche au sein du constructeur de la classe dérivée d'initialiser les attributs de la superclasse directement, puisque sauf si ceux-ci sont privés, vous y avez accès directement. L'appel implicite au constructeur de la superclasse risquerait de faire double emploi ou d'écraser vos valeurs.

2. Les destructeurs

Tout comme le constructeur est appelé à la création d'un objet, le destructeur est appelé quand l'objet est détruit ou l'allocation dynamique libérée. Tous les langages ne proposent pas de destructeurs, notamment Java, mais proposent parfois des méthodes particulières appelées automatiquement quand l'objet n'est plus utilisé (sortie de fonction quand l'objet est local, remise à NIL de sa valeur, etc).

PHP 5 propose un destructeur, appelé __destruct.

Le destructeur est rarement employé en algorithmique car les professeurs qui l'enseignent ont souvent le langage Java à l'esprit comme langage d'implémentation et pas le C++ par exemple. Pourtant il peut être important dans des langages où l'allocation de la mémoire est dynamique. Un attribut d'un objet peut être un pointeur alloué dynamiquement. Quand vous n'avez plus besoin de l'objet et que vous le libérez, que se passe-t-il pour ce pointeur ? Il faut libérer aussi la zone allouée.

Le destructeur ressemble beaucoup au constructeur, sauf que :

- Il n'y a qu'un seul destructeur.
- Il ne prend pas de paramètres.
- Son nom est composé d'un tilde suivi du nom de la classe.
- Vous le faites précéder du mot Destructeur.

Il dispose des mêmes propriétés pour les classes dérivées. Vous devez éventuellement l'appeler vous-même, en cascade.

```
Classe B hérite de A
  ...
```

```
  Constructeur B()
  Destructeur ~B()
  Début
    // Appel explicite du destructeur de A
    super.~A()
  Fin
FinClasse

Programme obj11
Var
  p_o1 :pointeur sur B
Début
  p_o1←nouveau B // appel implicite au constructeur
  ...
  Libérer p_o1 // appel implicite au destructeur
Fin
```

3. Les membres statiques

a. Attributs

Quand vous créez une instance de classe (un objet), chaque objet résultant dispose de son propre espace mémoire et chaque attribut donné dispose donc de sa propre valeur totalement indépendante de celle du même attribut d'un autre objet.

Pourtant il peut y avoir des cas où vous voudrez partager un attribut commun à tous les objets de ce type de classe. Le cas d'école est celui où vous voulez connaître le nombre de fois où la classe a été instanciée. Dans le cas de la classe Ecran, vous voudriez par exemple savoir combien d'objets de type Ecran ont été créés.

Pour cela, l'attribut doit être commun à tous les objets et chaque objet doit pouvoir modifier la valeur de cet attribut, valeur qui sera ensuite la même pour tous les objets. C'est le rôle de l'attribut **statique**.

Un attribut statique est un attribut qui n'est créé qu'en un seul exemplaire et qui est commun à tous les objets de la classe.

Vous le déclarez comme n'importe quel attribut, sauf que vous lui ajoutez le mot-clé "statique" après son type. L'attribut statique peut être privé ou public. Il peut être important d'initialiser une valeur par défaut pour un attribut statique.

```
Classe Ecran
  attributs privés
    nb_ecrans←0 :entier statique
    ...
FinClasse
```

Plutôt que d'utiliser **this** pour y accéder, vous utilisez le nom de la classe elle-même, tant depuis l'extérieur de l'objet (si l'attribut est public) que de l'intérieur. L'exemple suivant montre les possibilités associées avec deux objets : le constructeur ajoute 1, le destructeur soustrait 1. Les étapes intermédiaires affichent l'état de l'attribut statique nb_ecrans.

```
Classe Ecran
  attributs publics
    nb_ecrans←0 :entier statique
    ...
  méthodes publiques
    Constructeur Ecran()
    Début
      nb_ecrans←nb_ecran+1
    Fin
    Destructeur ~Ecran()
    Début
      nb_ecrans←nb_ecran-1
    Fin
    Procédure nbEcrans()
    Début
      Afficher nb_ecrans
  FinProc
FinClasse
```

```
Programme obj11
Var
  p_o1 :pointeur sur Ecran
  p_o2 :pointeur sur Ecran
Début
  p_o1←nouveau Ecran
  p_o1->nbEcrans() // 1

  p_o2←nouveau Ecran
  p_o2->nbEcrans() // 2

  Afficher Ecran.nb_ecrans // Accès public : 2

  Libérer p_o1
  p_o2->nbEcrans() // 1

  Libérer p_o2
  Afficher Ecran.nb_ecrans // Seul moyen ici : 0
Fin
```

4. Classes et méthodes abstraites

Il y a des fois où vous avez besoin de définir un prototype de classe dans laquelle vous allez déclarer des méthodes, mais pas forcément leur implémentation. Par exemple, vous voulez créer des classes représentant des figures géométriques afin de concevoir un logiciel de conception assistée par ordinateur. Dans ces logiciels (2D ou 3D), les différentes figures tracées ont toutes leurs propriétés comme leurs dimensions, surface, périmètre, volume, couleur de remplissage, couleur du contour, etc. Les valeurs de ces attributs varient d'une figure à l'autre, c'est normal, mais certaines méthodes peuvent être identiques comme celles pour la couleur et d'autres non comme le calcul des surfaces, volumes et périmètres. Pourtant toutes les figures (carrés, rectangles, polygones, triangles, trapèzes, losanges, etc) doivent disposer de ces méthodes. Comment décrire ceci en objet ?

La solution consiste à écrire une classe de base qui deviendra la superclasse de toutes les figures géométriques et qui va contenir tous les attributs de base comme la position de départ x,y du tracé de la figure, ses couleurs, mais aussi toutes les méthodes non seulement de base mais qui doivent aussi obligatoirement être implémentées dans toutes les classes qui en dérivent. Les méthodes de calcul de surface et de périmètre doivent être déclarées dans cette classe. Pourtant elles ne contiendront rien comme code : ce sera à la classe dérivée de les programmer.

Cette classe de base contenant des méthodes sans implémentation ne pourra pas être instanciée, vous ne pourrez pas créer d'objet à partir d'elle, car elle est de fait inutilisable et ne sert que de base pour les méthodes dérivées.

Une classe non instanciable et contenant des méthodes non implémentées, uniquement destinée à être dérivée, est appelée une classe abstraite. Les méthodes non implémentées qu'elle contient sont appelées méthodes abstraites.

- Pour créer une classe abstraite, vous ajoutez le mot-clé "abstraite" après le nom de la classe.
- Pour créer une méthode abstraite, vous ajoutez le mot-clé "abstraite" avant son nom.

```
Classe figure abstraite
  attributs privés
    x :réel
    y :réel
    c_contour :chaîne // couleur contour
    c_remp :chaîne :: couleur remplissage
    surface :réel
    périmètre :réel
  méthodes publiques
    Procédure setcolor_c(E :couleur :chaîne)
    Début
      this.c_contour←couleur
    Fin
    ...
    Fonction abstraite getSurface() :reel
    Fonction abstraite getPerimètre() :reel
    ...
FinClasse
```

À partir du moment où une classe contient une méthode abstraite, elle est elle-même abstraite. La classe figure est abstraite : elle contient des méthodes abstraites, sans implémentation. Elle contient aussi des méthodes qui ne sont pas abstraites comme setcolor_c() qui n'aura pas forcément à être réimplémentée dans les classes dérivées.

Il est de ce fait impossible de créer un objet de type figure. Vous devez maintenant créer les classes qui en héritent. Voici deux exemples simplifiés de classes : disque et rectangle. Vous devez implémenter dedans les méthodes abstraites de la superclasse.

Note importante : vous êtes **obligé** d'implémenter dans la classe dérivée les méthodes abstraites de la superclasse. Si vous ne souhaitez pas le faire, vous devez de nouveau déclarer la méthode comme abstraite et la classe dérivée sera elle-même abstraite.

```
Classe disque hérite de figure
  attributs privés
    rayon :réel
  méthodes publiques
    Constructeur disque(c_x,c_y,r)
    Début
      this.x←c_x
      this.y←c_y
      this.rayon←r
    Fin
    Fonction getSurface() :réel
    Début
      Retourne PI*this.rayon*this.rayon
    FinFonc
    Fonction getPerimètre() :réel
    Début
      Retourne 2*PI*this.rayon
    Fin
FinClasse

Classe rectangle hérite de figure
  attributs privés
    largeur :réel
    hauteur :réel
  méthodes publiques
    Constructeur rectangle(c_x,c_y,r_l,r_h)
    Début
      this.x←c_x
      this.y←c_y
      this.largeur←r_l
      this.hauteur←r_h
    Fin
    Fonction getSurface() :réel
    Début
      Retourne this.largeur*this.hauteur
    FinFonc
    Fonction getPerimètre() :réel
    Début
      Retourne 2*(this.largeur+this.hauteur)
    Fin
FinClasse
```

Les deux classes disque et rectangle ne comportant pas de méthodes abstraites, elle ne sont pas abstraites et peuvent être instanciées.

5. Interfaces

Les classes abstraites sont des classes comportant des méthodes implémentées ou non. Elles servent généralement de superclasses à d'autres. Il est possible de pousser le raisonnement plus loin : qu'est-il nécessaire de faire pour créer une classe servant uniquement et entièrement de prototype à une classe ? Cette classe doit avoir les propriétés suivantes :

- Non instanciable

- Abstraite

- Ne contenant que des méthodes abstraites.

Ainsi toute classe dérivant de cette classe spéciale devrait obligatoirement implémenter toutes les méthodes. Quel est l'intérêt ? Celui de définir un modèle unique et complet de méthodes pour les classes qui décident de les utiliser.

Ces types particuliers de classes s'appellent des **interfaces**. En pratique une classe n'hérite pas d'une interface : mis à part les définitions des méthodes une interface ne contient pas de code. Pour déclarer une classe, vous utilisez le mot-clé Interface à la place de classe. Par exemple, vous voulez créer une interface qui déclare toutes les méthodes de base pour la lecture d'un fichier multimédia : lecture, pause, stop, avance rapide, retour rapide, piste précédente, piste suivante. L'interface ressemblerait à ceci (les paramètres des méthodes ne sont pas précisés ici) :

```
Interface lecture
  méthodes publiques
    Procédure lecture() ;
    Procédure pause() ;
    Procédure stop() ;
    Procédure avance() ;
    Procédure retour() ;
    Procédure précédent() ;
    Procédure suivant() ;
FinInterface
```

 Notez bien qu'une interface ne contient QUE des méthodes et rien d'autre.

On dit qu'une classe qui décide d'utiliser une interface implémente les méthodes de l'interface, donc **implémente** l'interface. Vous utilisez ce même mot-clé pour le préciser.

```
Classe titi implémente interface
```

Vous pouvez décider d'implémenter plusieurs interfaces au sein de votre classe, dans ce cas vous séparez les noms des interfaces par des virgules.

```
Classe titi implémente interface1, interface2, interfacen
```

Vous devez implémenter toutes les méthodes de l'interface dans la classe. Si vous ne le faites pas, la méthode qui n'est pas implémentée doit être déclarée abstraite et la classe devient abstraite, donc non instanciable.

La classe Musique récupère l'interface lecture :

```
Classe Musique implémente lecture
  attributs privés
    morceau :chaîne
    piste :chaîne
    position : entier
    duree : entier
  méthodes publiques
    Constructeur Musique(m,p :chaînes)
    Début
      this.morceau←morceau
      this.piste←p
      position←0
      // Méthode de calcul :durée du morceau en secondes
      duree←duree_morceau(morceau)
    Fin
    // Début d'implémentation de l'interface
    Procédure lecture()
    Début
      ...
    FinProc
    Procédure pause()
    Début
      ...
    Fin
    Procédure stop()
    Début
```

```
        ...
    Fin
    // Continuez ici
    ...
FinClasse
```

L'objet en PHP

1. Les langages objet

Les deux parties précédentes de ce chapitre vous ont présenté ce qu'est la programmation objet. Il existe plusieurs langages orientés Objet. On parle de Programmation Orientée Objet (POO). Les prémices de la programmation objet sont anciennes (pour de l'informatique) : 1967 avec le langage Simula, puis Smalltalk. L'âge d'or débute au début des années 1980, avec tout d'abord l'Objective C (encore très utilisé notamment dans le développement de MacOS et de ses outils), puis le "C with Classes" en 1983, qui deviendra le C++, puis Eiffel, puis le Lisp Objet et ainsi de suite.

Les années 1990 ont vu l'explosion de l'objet à toutes les sauces pour le meilleur et pour le pire et pas uniquement dans les langages. Ainsi des bases de données objet sont apparues. D'anciens langages comme le Pascal, le Basic ou même le COBOL ont évolué vers l'objet : Delphi, Visual Basic, Cobol Objet, etc. Même les langages macros propres à des suites bureautiques (MS Office, OpenOffice.org) ou de base de données (Access, Paradox) se targuent d'être des langages objet. Si la définition de l'objet correspond à peu près à ce qui vous a été présenté depuis le début de ce chapitre, les puristes doivent dans certains cas bien rire.

L'arrivée de Java comme vrai langage objet de haut niveau a changé beaucoup de choses : reprenant les bonnes idées du C++, il l'adapte en un tout plus simple et plus pratique, adapté aux besoins modernes des développeurs et surtout des applications. La force de SUN a été de proposer un langage adapté à une large gamme de besoins : des applets Java, l'apprentissage de l'objet, les applications déportées, les serveurs d'applications, etc.

Un gros concurrent de la plate-forme Java (Java, ses outils, ses API, etc) se nomme .NET avec le langage C# (prononcez C sharp) et est évidemment objet. Les exemples de ce livre auraient parfaitement pu être développés en C#, sachant que le langage est maintenant disponible ailleurs que sous Windows.

Un autre concurrent de Java et de .NET, est le PHP. L'objet est apparu dans la version 4 de PHP, à la suite de nombreuses demandes en ce sens. Il était cependant très simple et plutôt limité. Le PHP est devenu un vrai langage objet avec la version 5. Son modèle est Java. Il est dans les faits assez simple de convertir les programmes Java en PHP.

La suite vous présente des notions d'objet en PHP. Elle se limitera aux notions que vous avez rencontrées depuis le début de ce chapitre.

2. Déclaration des classes et objets

Inconsciemment ou non, vous utilisez l'objet depuis la présentation des structures en PHP pour lesquelles il a été question de "tricher". Les structures programmées en PHP sont des objets.

Une classe, ses attributs et ses membres se déclarent ainsi :

```
class nom_classe {
  // liste des attributs
  public $att1 ;
  protected $att2 ;
  private $att3 ;

  // liste des méthodes
  public function methode1() {
    // code méthode1
  }
  private function methode2() {
    // code méthode2
  }
}
```

Vous remarquez que cela ressemble très fortement aux structures du chapitre Les tableaux et structures. Et pour cause, il a fallu tricher en PHP pour utiliser une classe comme une structure. La structure est en fait une classe uniquement composée d'attributs publics, d'où l'expression courante qui dit qu'une classe est une structure avec des fonctions.

Voici un exemple simplifié d'implémentation de la classe Ecran faisant un constructeur avec trois paramètres passés à la valeur null par défaut et une méthode `modif_type()` appelée par l'un des constructeurs ou d'où vous voulez, puisque la méthode est publique. Cet exemple fait aussi apparaître l'utilisation d'une variable statique et son utilisation depuis le programme principal.

```
<html>
```

```
  <head><meta/>
    <title>Objets Ecran</title>
  </head>
  <body>
  <?php

class Ecran {
  private $type,$marque,$modele,$connecteur;
  private $resolution;
  private $diagonale;
  private $hauteur,$largeur,$profondeur,$poids;
  public static $nb_ecrans=0;

  // constructeur par défaut

  // constructeur avec 3 arguments
  function Ecran($t=null, $mq=null, $mdl=null) {
    echo "Constructeur avec trois arguments<br />";

    if($t==null) {
      $this->type="Inconnu" ;
      $this->marque="Inconnu" ;
      $this->modele="Inconnu" ;
      $this->nb_ecrans++ ;
    } else {
      $this->modif_type($t) ;
      $this->marque=$mq ;
      $this->modele=$mdl;
      self::$nb_ecrans++ ;
    }
  }

  public function modif_type($mod) {
    $tmod=array("CRT","LCD","PLASMA");

    $ok=array_search($mod,$tmod);

    if($ok) $this->type=$mod;
    return $ok;
  }

  public function affiche_type() {
    echo $this->type."<br />";
  }

  // fonction statique
  public static function nbEcrans() {
    echo self::$nb_ecrans."<br />";
  }
}

  // allocation dynamique avec construction
  $o1=new Ecran("LCD","LG","L1915S");

  // appel d'une méthode
  $o1->affiche_type();

  // Les deux lignes font la même chose
  Ecran::nbEcrans();
  echo $o1->nbEcrans()."<br />";

?>
  </body>
</html>
```

Contrairement à l'algorithmique, les attributs et méthodes ne sont pas regroupés par blocs publics, privés ou protégés. C'est à vous d'ajouter avant le nom et le type du membre s'il est public, privé ou protégé. Les définitions public et **private** sont exactement celles vues ci-dessus, tandis que **protected** limite la portée des attributs aux classes dérivées.

Il n’est pas possible d’utiliser le polymorphisme en PHP, ce qui empêche la création de plusieurs constructeurs. Il faut donc ruser en conséquence. Ici c’est assez simple, mais il existe d’autres moyens de vérifier la liste des paramètres passés à une fonction.

3. Héritage

PHP autorise l’héritage simple mais pas l’héritage multiple. Vous verrez cependant qu’il est possible de ruser un petit peu avec les interfaces. L’exemple des figures géométriques peut s’appliquer ici, car PHP supporte très bien les classes et méthodes abstraites.

L’héritage est exactement comme en algorithmique, sauf qu’à la place de "hérite de" vous utilisez le mot-clé **extends**. Concernant les classes et méthodes abstraites, vous ajoutez devant le nom le mot-clé **abstract**.

Notez que dans la définition de la classe abstraite, quelques attributs privés sont remplacés par des attributs protégés. Autrement, les classes dérivées n’auraient pas eu le droit d’y accéder.

Enfin notez aussi l’utilisation du mot-clé **parent** qui remplace la classe (ou plutôt l’objet) héritée, utilisé ici pour appeler le constructeur de la classe figure chargé de placer les bonnes couleurs.

```
<html>
  <head><meta/>
    <title>Heritage en PHP</title>
  </head>
  <body>
  <?php

  abstract class figure {
    protected $x,$y,$surface,$perimetre;
    protected $c_contour,$c_remp;
    const PI=3.1415927;

    function figure($c1, $c2) {
      $this->setcolor_c($c1);
      $this->setcolor_r($c2);
    }

    public function setcolor_c($couleur) {
      $this->c_contour=$couleur;
    }

    public function setcolor_r($couleur) {
      $this->c_remp=$couleur;
    }

    abstract public function getSurface();
    abstract public function getPerimetre();
  }

  class disque extends figure {
    private $rayon=0;

    // constructeur
    function disque($c_x, $c_y, $r) {
      parent::__construct("noir","blanc");
      $this->x=$c_x;
      $this->y=$c_y;
      $this->rayon=$r;
    }

    // implémentation des méthodes abstraites
    public function getSurface() {
      return self::PI*$this->rayon*$this->rayon;
    }

    public function getPerimetre() {
      return 2*self::PI*$this->rayon;
    }
  }
```

```
  class rectangle extends figure {
    private $largeur, $hauteur;

    // constructeur
    function rectangle($c_x,$c_y,$r_l,$r_h) {
    parent::__construct("violet","marron");
    $this->x=$c_x;
    $this->y=$c_y;
    $this->largeur=$r_l;
    $this->hauteur=$r_h;
  }

  // implémentation des méthodes abstraites
  public function getSurface() {
    return $this->largeur*$this->hauteur;
  }

  public function getPerimetre() {
    return 2*($this->largeur+$this->hauteur);
  }
}

  $o1=new disque(100,100,10);
  echo $o1->getSurface()."<br />";
  echo $o1->getPerimetre()."<br />";

  $o2=new rectangle(10,10,35,25);
  echo $o2->getSurface()."<br />";
  echo $o2->getPerimetre()."<br />";

?>
  </body>
</html>
```

4. Interfaces

Encore une fois, en PHP les interfaces sont exactement comme en algorithmique. Vous créez une interface avec le mot-clé **interface** et vous indiquez que la classe implémente cette interface avec le mot-clé **implements** après le nom de la classe, suivi du nom de l'interface.

```
<html>
  <head><meta/>
    <title>Interfaces en PHP</title>
  </head>
  <body>
  <?php


  interface lecture {
    public function lecture() ;
    public function pause() ;
    public function stop() ;
    public function avance() ;
    public function retour() ;
    public function precedent() ;
    public function suivant() ;
  }

  class Musique implements lecture {
    private $morceau;
    private $position,$duree,$piste;

    function Musique($m, $p) {
      $this->morceau=$m;
      $this->piste=$p;
      $position=0;
    }
```

```
    // Début d'implémentation de l'interface
    public function lecture() {
      echo "lecture de {$this->morceau}<br />";
    }
    public function pause() {
      echo "Pause à {$this->position}<br />";
    }
    public function stop() {
      echo "Arrêt piste {$this->piste}<br />";
    }
    public function avance() {
      echo "Avance<br />";
    }
    public function retour() {
      echo "retour<br />";
    }
    public function precedent() {
      echo "Précédent<br />";
    }
    public function suivant() {
      echo "Suivant<br />";
    }
  }

  $o1=new Musique("Somebody to love",10);
  $o1->lecture();
?>
  </body>
</html>
```

Une classe peut implémenter plusieurs interfaces. Dans ce cas, séparez les noms des interfaces par des virgules.

Une interface peut hériter d'une ou plusieurs autres interfaces via le mot-clé **extends** suivi du nom de la ou des interfaces héritées.

```
  interface lecture1 {
    public function lecture() ;
    public function pause() ;
    public function stop() ;
  }
  interface lecture2 {
    public function avance() ;
    public function retour() ;
    public function precedent() ;
  }
  interface lecture extends lecture1, lecture2 {
    public function suivant() ;
  }
```

Ces deux derniers cas sont les seuls en PHP où, indirectement, une sorte d'héritage multiple a été mis en place, dans le sens où une classe peut hériter de plusieurs définitions de méthodes. C'est simpliste dans le sens où de toute façon, ces méthodes doivent être implémentées obligatoirement dans la classe qui utilise les interfaces et qu'ainsi, PHP n'a pas à savoir s'il doit utiliser telle ou telle méthode de l'une des classes héritées.